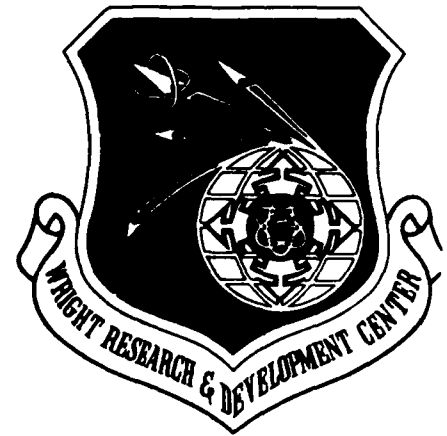


DTIC FILE COPY

AD-A217 730

RDC-TR-89-3114

LABORATORY IMPLEMENTATION OF THE CONTINUOUSLY RECONFIGURING  
MULTI-MICROPROCESSOR FLIGHT CONTROL SYSTEM (CRMMFCS)



Bill Rollison, Dan Thompson, Ray Bortner,  
Dan Pruett, Mark Mears  
Control Systems Development Branch  
Flight Control Division

October 1989

Final Report for Period April 1980 - June 1984

Approved for Public Release; Distribution is Unlimited

DTIC  
ELECTE  
FEB 07 1990  
S E D

LIGHT DYNAMICS LABORATORY  
WRIGHT RESEARCH AND DEVELOPMENT CENTER  
AIR FORCE SYSTEMS COMMAND  
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6553

90 02 07 056

NOTICE

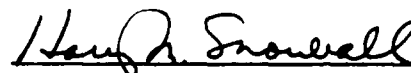
When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.



WILLIAM E. ROLLISON  
Project Manager  
Control Systems Development Branch  
Flight Control Division

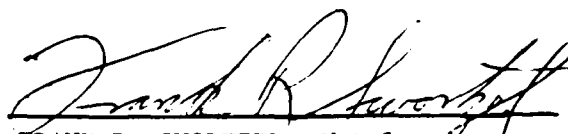


HARRY M. SNOWBALL, Tech. Grp. Mgr.  
Control Data Group  
Control Systems Development Branch  
Flight Control Division

FOR THE COMMANDER



H. MAX DAVIS  
Assistant For Research & Technology  
Flight Control Division



FRANK R. SWORTZEL, Chief  
Control Systems Development Branch  
Flight Control Division

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WRDC/FICLB, WPAFB, OH 45433-6553 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

## REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

1a. <del>PROPOSED</del> SECURITY CLASSIFICATION			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for Public Release; Distribution is Unlimited		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S)  WRDC-TR-89-3114			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION  Flight Dynamics Laboratory		6b. OFFICE SYMBOL (if applicable) WRDC/FIGLB		7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code) Flight Dynamics Laboratory (WRDC/FIGLB) Wright Research and Development Center Wright-Patterson AFB OH 45433			7b. ADDRESS (City, State, and ZIP Code)		
8a. NAME OF FUNDING / SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (if applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State, and ZIP Code)			10. SOURCE OF FUNDING NUMBERS		
			PROGRAM ELEMENT NO. 62201F	PROJECT NO. 2403	TASK NO. 02
			WORK UNIT ACCESSION NO. 44		
11. TITLE (Include Security Classification) Laboratory Implementation of the Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMMFCS)					
12. PERSONAL AUTHOR(S) Rollison, Thompson, Bortner, Pruett, Mears					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM April 1980 TO June 1984		14. DATE OF REPORT (Year, Month, Day) 1989 October	
15. PAGE COUNT 120					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Digital Control Systems, Multi-Microprocessor, Flight Control Systems, Distributed Control, Reconfiguration, Distributed Systems		
17	07				
12	07				
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>The initial Continuously Reconfiguring Multi-Microprocessor Flight Control System [CRMMFCS] report [AFWAL-TR-81-3070] highlighted the theoretical concepts and established the boundaries for realizing an autonomously distributed control system. This report covers the hardware and software methods used in the laboratory model to achieve the goals as set down in the concept phase. The discussion highlights the hardware construction, revealing the operations and the reasoning behind the choices made. The section on software reviews the techniques used to glue the hardware together to become a system. The completed laboratory model and its testing under a number of conditions are discussed.</p>					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified		
22a. NAME OF RESPONSIBLE INDIVIDUAL William E. Rollison			22b. TELEPHONE (Include Area Code) (513) 255-8293		22c. OFFICE SYMBOL WRDC/FIGLB

# TABLE OF CONTENTS

PAGE

1	Introduction.....	1
2	CRMMFCS Concepts.....	2
2.1	Introduction.....	2
2.2	Continuous Reconfiguration.....	2
2.3	Autonomous Control.....	5
2.4	Virtual Common Memory.....	6
2.5	Transparent Bus Contention.....	9
2.6	Fault Filter.....	10
3	Progress Report of CRMMFCS LAB Implementation.....	12
4	Implementation Description.....	18
4.1	Hardware.....	18
4.1.1	Introduction.....	18
4.1.2	Microprocessor Selection.....	22
4.1.3	Bus Structure Selection.....	23
4.1.4	Bus Access.....	23
4.1.5	CRMMFCS Bus Specifics.....	25
4.1.6	Bus Termination and Synchronization.....	25
4.1.7	Transmitter.....	28
4.1.8	Receiver.....	39
4.2	Software.....	43
4.2.1	Introduction.....	43
4.2.2	Development Facilities and Initial Decisions.....	44
4.2.3	Development Support Software.....	48
4.2.4	Typical Multiprocessor Software Problem.....	53
4.2.5	The CRMMFCS Operating System.....	55

# TABLE OF CONTENTS

PAGE

4.2.5.1	Control Requirements.....	55
4.2.5.2	Modular Structure.....	56
4.2.5.3	OS Kernel.....	56
4.2.5.4	Millisecond Interrupt and Subtask Synchronization.....	59
4.2.5.5	Communications Hardware from Software Prospective.....	60
4.2.5.6	Tasking and Reconfiguration.....	61
4.2.5.7	Implementation Time Divisions.....	71
4.2.5.8	Fault Tolerance.....	71
4.2.6	Application Software: The Control Law Model.....	80
4.2.7	Simulation Software: MC 68000.....	85
4.2.8	Other Software Developed.....	87
4.3	Implementation Laboratory Demonstration.....	93
5	Implementation Performance Data.....	96
5.1	Throughput.....	96
6	Problems Identified and Possible Corrections.....	103
6.1	Corrections.....	103
6.2	Other Identified Problems.....	111
7	Updates to CRMMFCS if a Redesign was Attempted.....	112
8	Further Areas of Investigations.....	115
8.1	Benefits of CRMMFCS and Related Research.....	115
8.2	Other Issues.....	117
9	Conclusions.....	118
10	Bibliography.....	120

# LIST OF FIGURES

PAGE

FIGURE 1. Reconfiguration Example.....	4
FIGURE 2. Volunteering Table.....	7
FIGURE 3. Virtual Common Memory Concept.....	8
FIGURE 4. Transparent Bus Contention.....	11
FIGURE 5. Fault Filter.....	13
FIGURE 6. Hardware Milestone Chart.....	14
FIGURE 7. Software Milestone Chart.....	15
FIGURE 8. Breadboard of Receiver and Transmitter.....	16
FIGURE 9. First Multiple Module Test Configuration.....	17
FIGURE 10. Multiple Module Test Run Configuration.....	19
FIGURE 11. TMS 9900 Microcomputer Board.....	20
FIGURE 12. TMS 9900 Dynamic Ram Board.....	21
FIGURE 13. Bus Clock and Termination Circuit.....	27
FIGURE 14. Transmitter Printed Circuit Board.....	29
FIGURE 15. Transmitter Bus Interface.....	30
FIGURE 16. Transmitter Block Diagram.....	32
FIGURE 17. Buss Model.....	35
FIGURE 18. Transmission Format.....	36
FIGURE 19. Bus Access Gate (BAG) Block Diagram.....	38
FIGURE 20. Logic Analyzer Diagram of Sync Pulses.....	40
FIGURE 21. Receiver Printed Circuit Board.....	41
FIGURE 22. Receiver Block Diagram.....	42
FIGURE 23. 8002 Development System.....	45
FIGURE 24. 8002 Interface to CRMMFCS Buss.....	46
FIGURE 25. Task Assignment Tree.....	57
FIGURE 26. Operating System Kernel Flowchart.....	58
FIGURE 27. Address Word Formatting.....	62
FIGURE 28. TMS 9900 Assembly Language Transmission Formatting Example.....	63
FIGURE 29. Volunteer Status Table and Process.....	66
FIGURE 30. Required Latency of Scheduling of Subtasks.....	70

LIST OF FIGURES	PAGE
FIGURE 31. Laboratory Implementation of Time Divisions.....	72
FIGURE 32. Blackmark Table.....	75
FIGURE 33. Control Laws Block Diagram.....	82
FIGURE 34. Matrix Form of Control Equations.....	83
FIGURE 35. Separation of Control Law Elements.....	84
FIGURE 36. MC 68000 Computer Board.....	86
FIGURE 37. MC 68000 Interface to CRMMFCS Buss.....	88
FIGURE 38. Matrix Form of Airframe Simulation Equations.....	89
FIGURE 39. Laboratory Displays' Interface to CRMMFCS Buss.....	91
FIGURE 40. Volunteering and Blackmark Table.....	92
FIGURE 41. Systems' Laboratory Implementation.....	94
FIGURE 42. Reconfiguration Overhead.....	99
FIGURE 43. Minimum Reconfiguration Overhead.....	100
FIGURE 44. Compound Millimodules.....	102

## DEDICATION

We would like to use this space to recognize a member of the Wright Development and Research Center's Flight Dynamics Laboratory team. ELISHA RACHOVITSKY. Rocky, as he was known to us, died in November of 1988. He will be missed, because not only was he a colleague but a friend. It is truly a privilege to say we knew Rocky.

<b>Accession For</b>	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	





1.

## Introduction

In late 1978 the WRDC Flight Dynamics Laboratory began a program to study the potential benefits of applying microprocessor technology to flight control. Research by members of the Control Systems Development Branch (WRDC/FIGL) of the Flight Control Division resulted in the initiation of an in-house development effort to produce a laboratory fault tolerant, multi-microprocessor flight control system. In May 1981, a technical report (TR) was released describing the Continuously Reconfiguring Multi-Microprocessor Flight Control System (CRMMFCS) concept. The report, AFWAL-TR-81-3070, was written by the concept originators, Capt. Stanley Larimer and Lt. Scott Maher, describing the efforts from 1 August 1979 to 30 April 1981. This period primarily covers the conceptual phase of the program. This report describes the implementation and analysis phases of the CRMMFCS program and covers the period from April 1980 to June 1984.

The original goal in the conceptual phase of the program is to study microprocessor applications in flight control. The research extended into the implementation phase of CRMMFCS. The intent is not to produce a flight-worthy, verified, final product system. The aim is to prove that multi-microprocessor systems can be used for fault tolerant flight control, and that the concepts envisioned for CRMMFCS can be applied to achieve these goals. A direct result of the implementation was the study of the issues of reliable multiprocessor systems first-hand, so that the state of the art in flight control systems can be advanced by the Air Force.

The report begins with a brief review of the main features of CRMMFCS. For additional details, see AFWAL-TR-81-3070. Next, a progress section outlines the major milestones of the project. The remaining

sections will describe the hardware and software, laboratory demonstration, performance data, problems discovered with the implementation, potential updates, and future areas of concern.

2.

## CRMMFCS Concepts

### 2.1 Introduction

CRMMFCS is a system of autonomous microprocessors which communicate via a set of serial busses. The system is fully decentralized to avoid single failure points. The concept was structured to exercise and test all processors, identify errors, and transparently remove failed processors. Tasks are continuously redistributed among functioning processors, without the use of a central controller. This section reviews the main concepts of CRMMFCS, described in detail in AFWAL-TR-81-3070. The areas to be covered are continuous reconfiguration, autonomous control, virtual common memory (VCM), transparent bus contention, and the fault filter.

### 2.2 Continuous Reconfiguration

In fault tolerant systems, some backup mode of operation must exist to cover failed processing units. When a processor or subsystem fails, another must take over its computational tasks. In multiprocessor systems, sparing, the use of extra processors to take over when primary processors fail, is a logical technique. In one case the spares, like the processors they replace, are fixed in their tasks. When a processor dedicated to a particular task fails, one of its dedicated spares takes over. This requires the removal of one processor from the system, and the "attachment" of another.

A more efficient means of sparing is to utilize a pool of processors which act as spares for the entire system. The spares are

capable of assuming the task of any failed processor. Two possibilities exist in such a system, cold sparing or hot sparing. In the case of "cold spares," the new processor must be loaded with the necessary code to accomplish the task. "Hot spares" already have the necessary code; they need only to be assigned the job.

CRMMFCS uses a variation of the "hot spare" method called continuous reconfiguration. Continuous reconfiguration is the dynamic redistribution of tasks among the functioning processors of the system. Spares are not kept idle, awaiting to be assigned a task when another fails, instead they are treated like other tasks of the system. When the tasks are distributed, the "spare" tasks are assigned to processors not assigned "normal" tasks. Because tasks are continuously redistributed, the "spare" tasks are moved around from processor to processor. None of the processors in the system is a dedicated spare. The process is best demonstrated by an example.

Refer to Figure 1. In the first time frame, processor 1 is assigned task B, processor 2 task D, processor 3 a "spare" task, and so on. Before the next time frame, reconfiguration takes place. Processor 1 now takes task A, processors 2 and 3 take "spare" tasks, processor 4 task F, and so on. Note that all tasks are still computed, including the three "spare" tasks.

Processors, upon failure, drop out of system operation transparently. Suppose that before the third time frame of Figure 1, processor 4 fails and drops out. As a result of the task selection process, all primary tasks are assigned to functional processors.

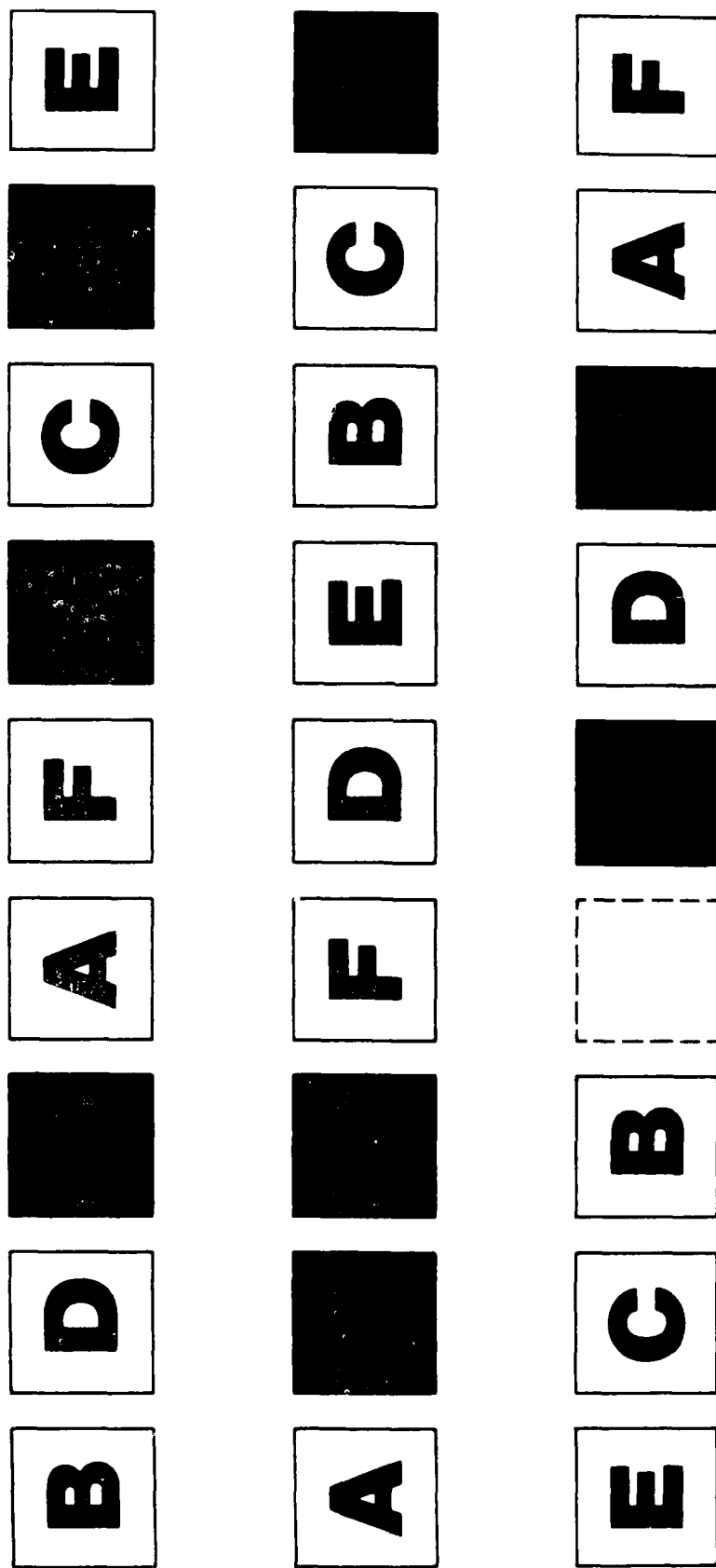


FIGURE 1. Reconfiguration Example

Processors left over after the primary tasks are selected are assigned "spare" tasks. Because of the one-to-one mapping of tasks to processors, failures in processors cause some tasks to go unperformed. Tasks are prioritized so that low priority tasks are the first to be lost. "Spare" tasks are considered the lowest priority, so they are dropped out first. In the third time frame of the example, processor 4 takes no task, and one "spare" task is not performed. The effect is transparent since the other processors in the system are not affected by the loss of the processor. In effect, only a "spare" is lost.

What are the advantages of continuous reconfiguration? The transparent removal of processors is certainly a primary benefit. No transient time is spent shutting down one processor and bringing up another. Loading the new processor with code or running a power-up self-test is not necessary. "Spares" in CRMMFCS are continually checked through use in normal system operation. In short, reconfiguration is not a condition entered only upon detection of a processor failure. In CRMMFCS, reconfiguration is the norm, not the exception.

### 2.3 Autonomous Control

CRMMFCS is a fully distributed system. A central controller is not utilized since this is vulnerable to a single failure point. CRMMFCS was designed as an autonomous system: no processor exerts control over another; each determines its own state. To accomplish autonomous control with continuous reconfiguration, several requirements must be satisfied: (1) the system must adhere to a well-defined set of task assignment rules, (2) the state information of the system must be available to all processors of the system, (3) each processor must have access to all the software (handled in CRMMFCS by simply giving each

processor an identical copy of the entire software suite), and (4) an efficient bus utilization scheme must be used.

The task assignment rules must not only be well-defined, but must also operate in a distributed manner. CRMMFCS incorporates a volunteering scheme that utilizes a table in the system state information that is accessible by all (see Figure 2) processors. Volunteering and choosing a task is accomplished according to the results of the state of that table. Volunteering in this case is defined as the act of a processor sending a healthy status message to a designated table location. In the event the processor does not send such a message or sends an "unhealthy" status value, the processor is unable to volunteer. Processors with valid entries in the table take tasks from a preordered list in the order of the table entries. Continuous reconfiguration is accomplished by moving around the logical starting point of the table. This logical starting point is indicated by the rotating offset pointer. For a more in-depth look at volunteering, see section 4.2.

#### 2.4 Virtual Common Memory (VCM)

The requirement that all state information of the system be accessible by all processors is met in CRMMFCS through the use of a shared memory. Since a single, physically shared memory raised reliability and access-bottleneck questions, a distributed shared memory was implemented. The Virtual Common Memory (VCM) concept is based on the premise that each processor has a local copy of the entire common memory. As Figure 3 shows, each processing unit has its own copy of the common memory (also called the state information matrix, or SIM). Transmissions sent by processors are received identically by all processing units and stored in their respective SIM's. Even though the SIM is local to the processing unit, writing to the memory still

# Volunteering Table

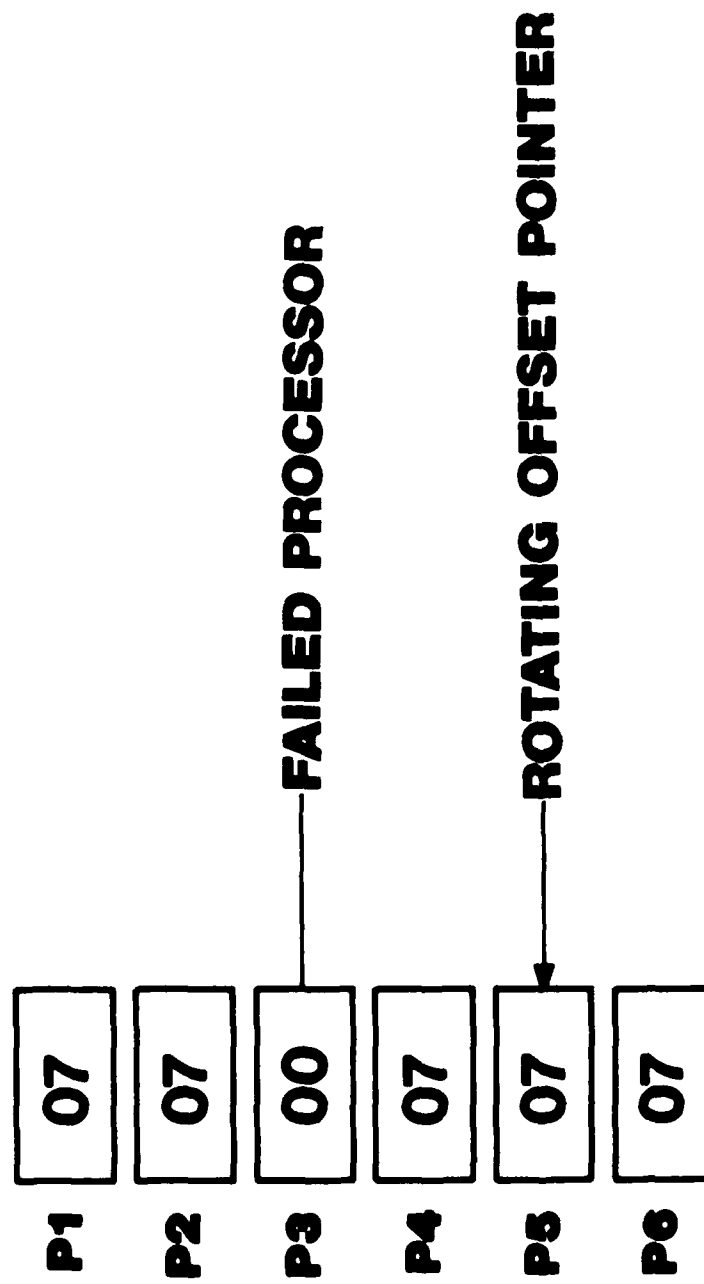


FIGURE 2. Volunteering Table

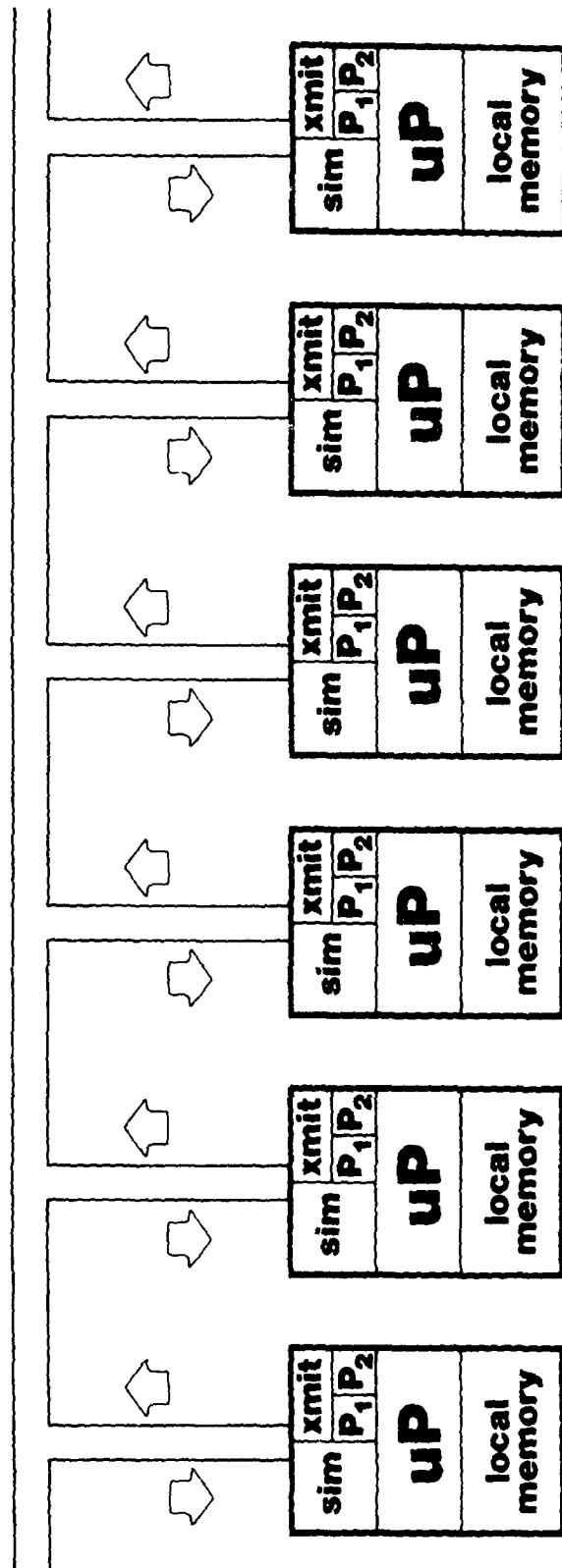


FIGURE 3. Virtual Common Memory Concept



requires a transmission. In this way, updates to the memory changes the state of the virtual memory so that all processors see the same data.

### 2.5 Transparent Bus Contention

The last requirement is an efficient bus utilization scheme. Efficiency is required so that high numbers of processors can be used in the system without "starving" some from adequate bus time. Access latency is an important consideration. Since a shared memory is being used, updates to state variables, such as the volunteering table, must be made in a timely manner. To minimize latency, individual transmissions are limited to one data item at a time. An access contention scheme is used to ensure that only processors that have a need to use a bus can have ownership of a bus.

A collision-type of contention scheme with open collector busses is utilized in CRMMFCS. Processors seek out a bus which is idle (logically high for a specified period of time). When one is found, the processor begins broadcasting its serial transmission. While transmitting, the processor reads back the state of the bus. If there is agreement between the data going out and that being read back, the transmission continues. A successful transmission is one that is sent completely and read back correctly.

Suppose that another processor also wants to use the same bus. If one starts before the other, the first processor to the bus gets the bus automatically since the bus no longer appears idle to the other processor. If they start at precisely the same time, contention takes place.

In contention, a processor will simply continue to transmit its data as long as it reads back exactly what it puts out. In a open collector format, logical lows "override" logical highs. Therefore, if

one processor puts out a low while another puts out a high, the low is what appears on the bus. In contention, then, processors lose the bus when they try to put out a logical high while another puts out a low. One processor always wins the contention; the others stop trying on that bus and move on to an idle bus.

Figure 4 gives an example. Two processors contend for a bus, and start exactly at the same time. The processors have the same data for the first two bits so they both remain in contention. At the third bit, however, processor (transmitter) one puts out a low while processor two puts out a high. The bus state in the open collector situation goes low; agreeing with processor one. Processor two will cease transmitting on this bus at this point. Processor one wins the contention, and continues on with its transmission. The contention was transparent since the processors were putting out real data to contend on. Overhead is not required for this type of contention.

## 2.6 Fault Filter

The last concept of CRMMFCS to be covered in this review is the fault filter. This is the logical view of how CRMMFCS catches, or filters, faults in the system. Faults can appear in various forms. They range from rare computation or transmission errors because of random noise or glitches to partial or total failure of processing elements or transmitting units. Given the spectrum of fault possibilities, the system of fault detection and isolation mechanisms must be able to cover a wide area. A single fault detection and isolation mechanism will not be able to cover this entire area.

CRMMFCS uses multiple fault detection and isolation mechanisms which overlap each other. In this way, failures which pass one or more types of tests is caught by at least one other. The goal is to utilize the right combination of "filter layers" necessary to catch all faults.

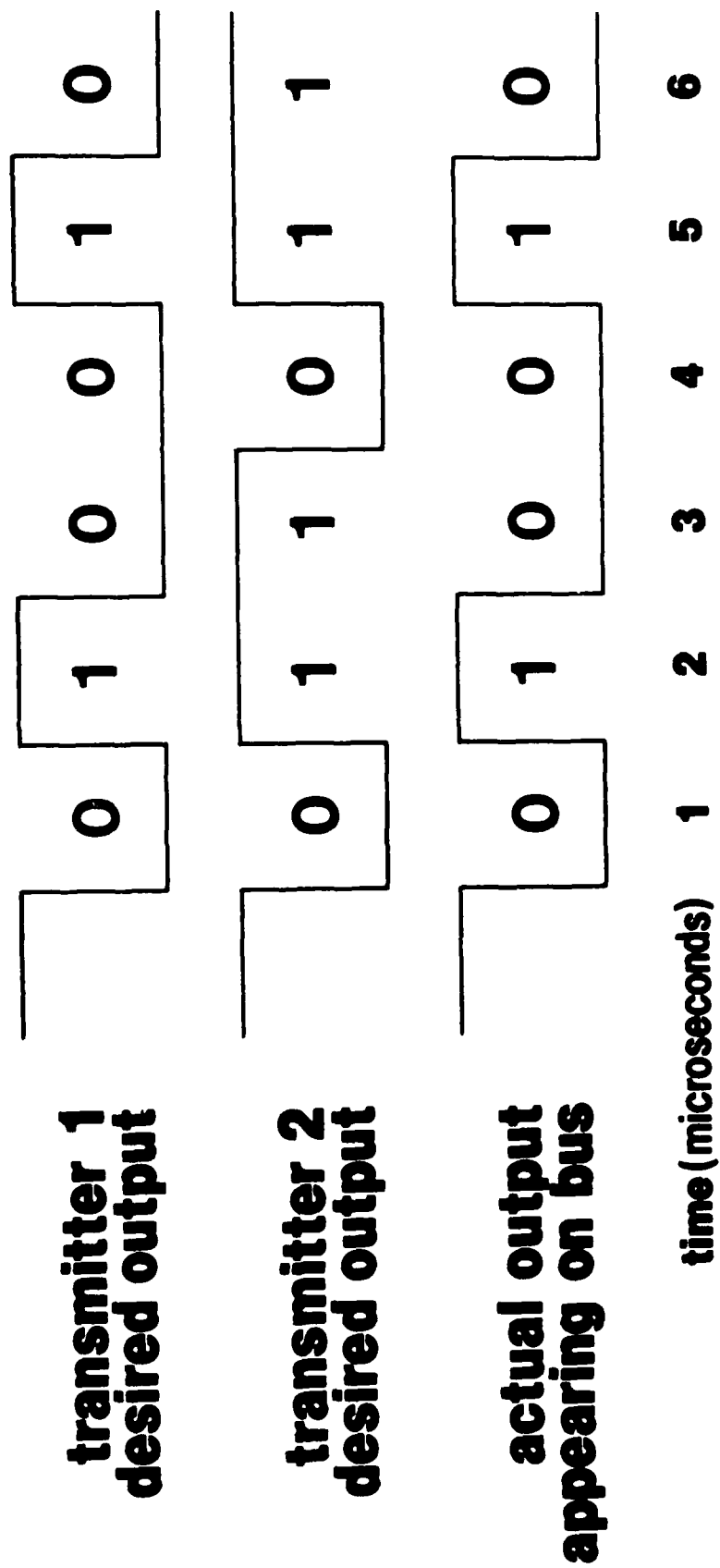


FIGURE 4. Transparent Bus Contention

In CRMMFCS, the main layers of the filter are given in Figure 5. The individual features will be described further in section 4.2 of this report. This report does not assume that the layers in Figure 5 are sufficient for a fault tolerant flight control system. The complexity of the filter depends upon the physical implementation and the desired level of reliability. The results of this CRMMFCS implementation will be covered in Section 5.

3

### Progress Report of CRMMFCS Laboratory Implementation

The laboratory implementation of CRMMFCS took approximately 2 ½ years to complete. Additional analysis work followed this for about 2 more years. Milestone charts are included in Figures 6 and 7.

The work performed in FY80 was mainly in initial planning and breadboarding. Aspects of the implementation, such as the chosen microprocessor and development tools, were discussed and studied during this time.

During FY81, the initial planning and breadboarding gave way to full-scale development. Software was added to test the single breadboard module in Figure 8 (processor/transmitter/receiver). Printed circuit(PC) boards were laid out for the transmitter and receiver boards once the major "bugs" were worked out of the breadboard versions. The first integration of multiple processing modules in a test configuration (Figure 9) was performed in the middle of FY81. The initial development work on the CRMMFCS operating system (OS) and control law model was also begun during this period.

By the beginning of FY82, construction of the PC boards had produced several operational sets. Each hardware module was tested in a stand-alone configuration to work out construction and failed component problems. The modules were then integrated into a multiple processing

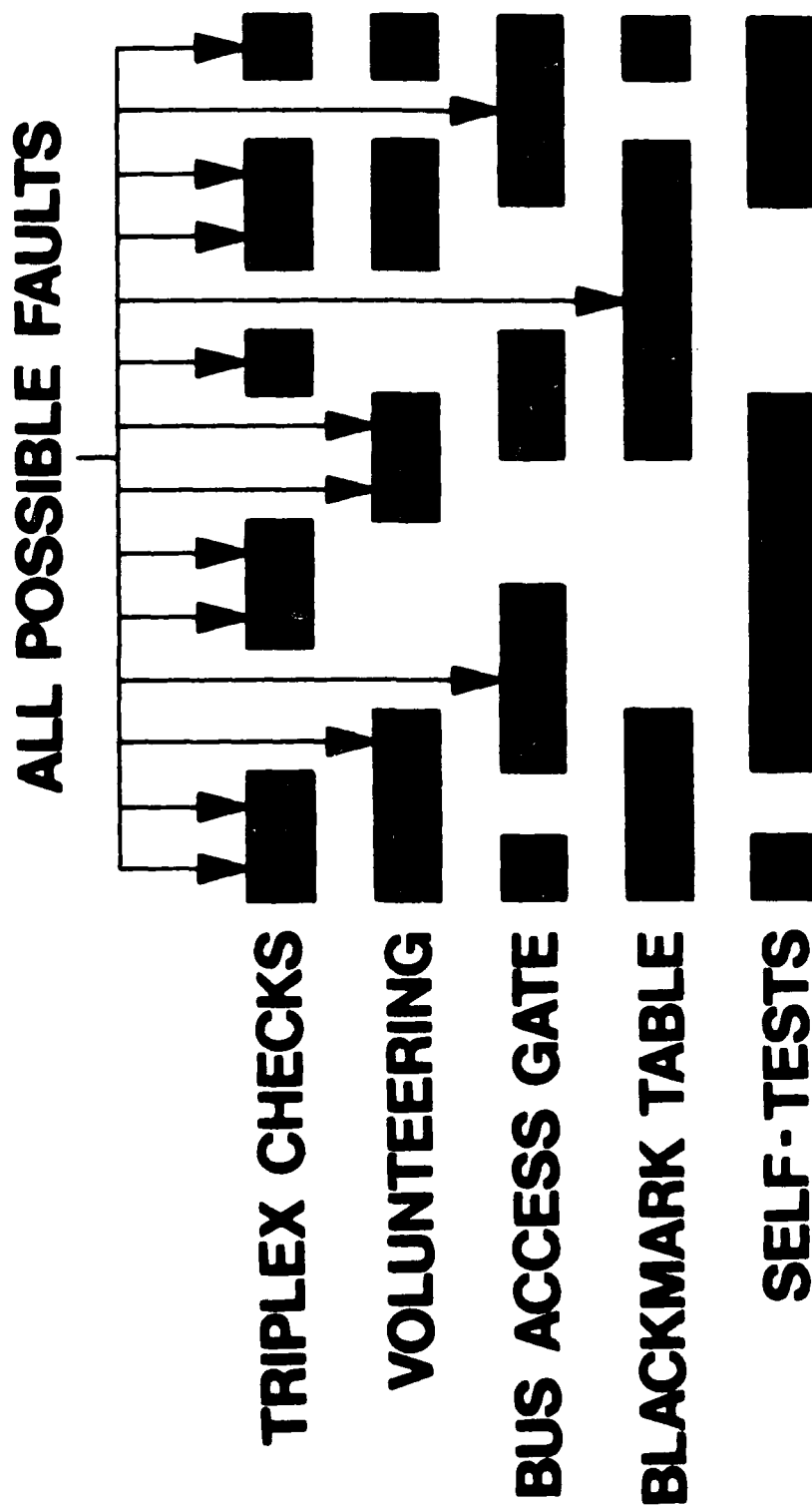
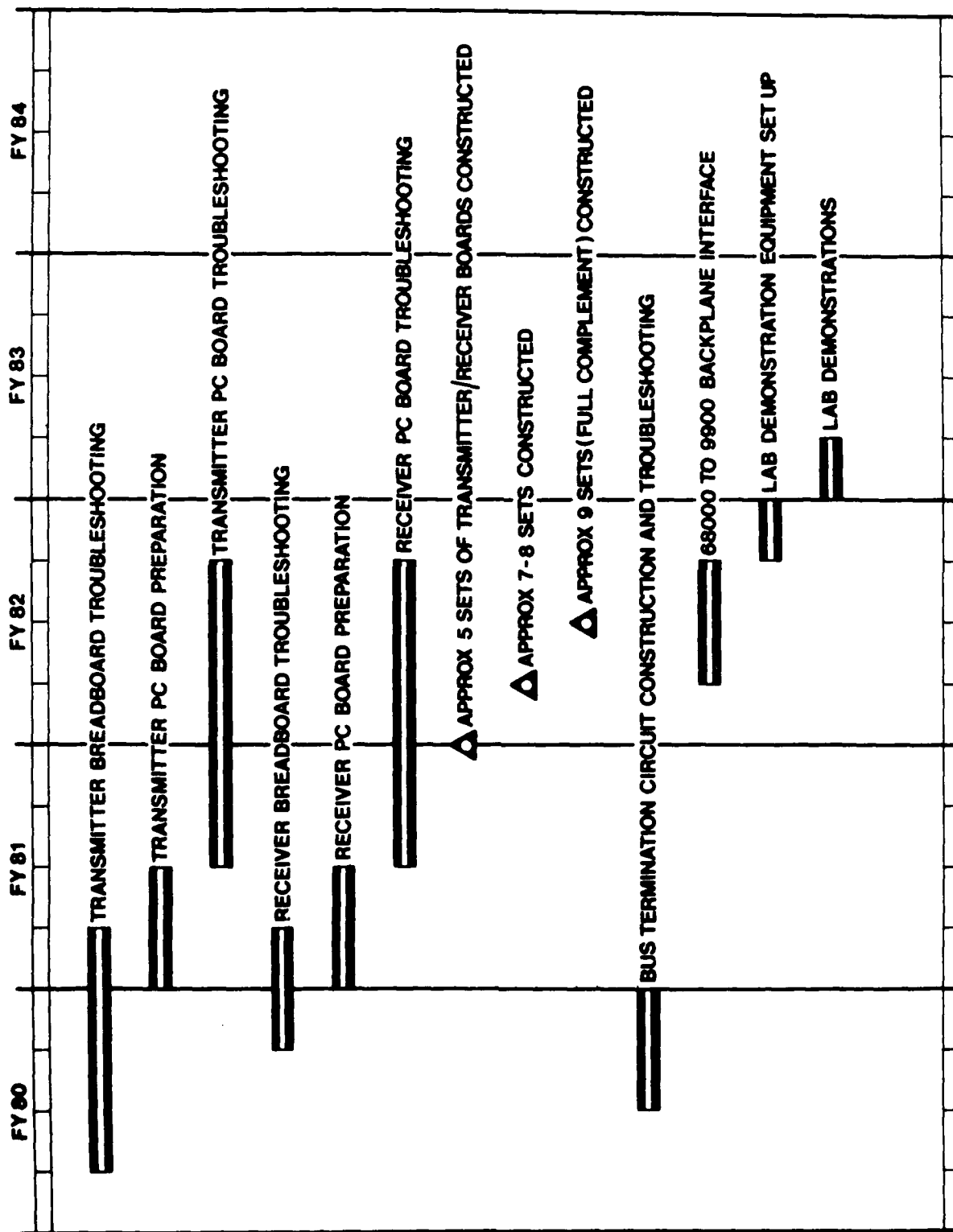
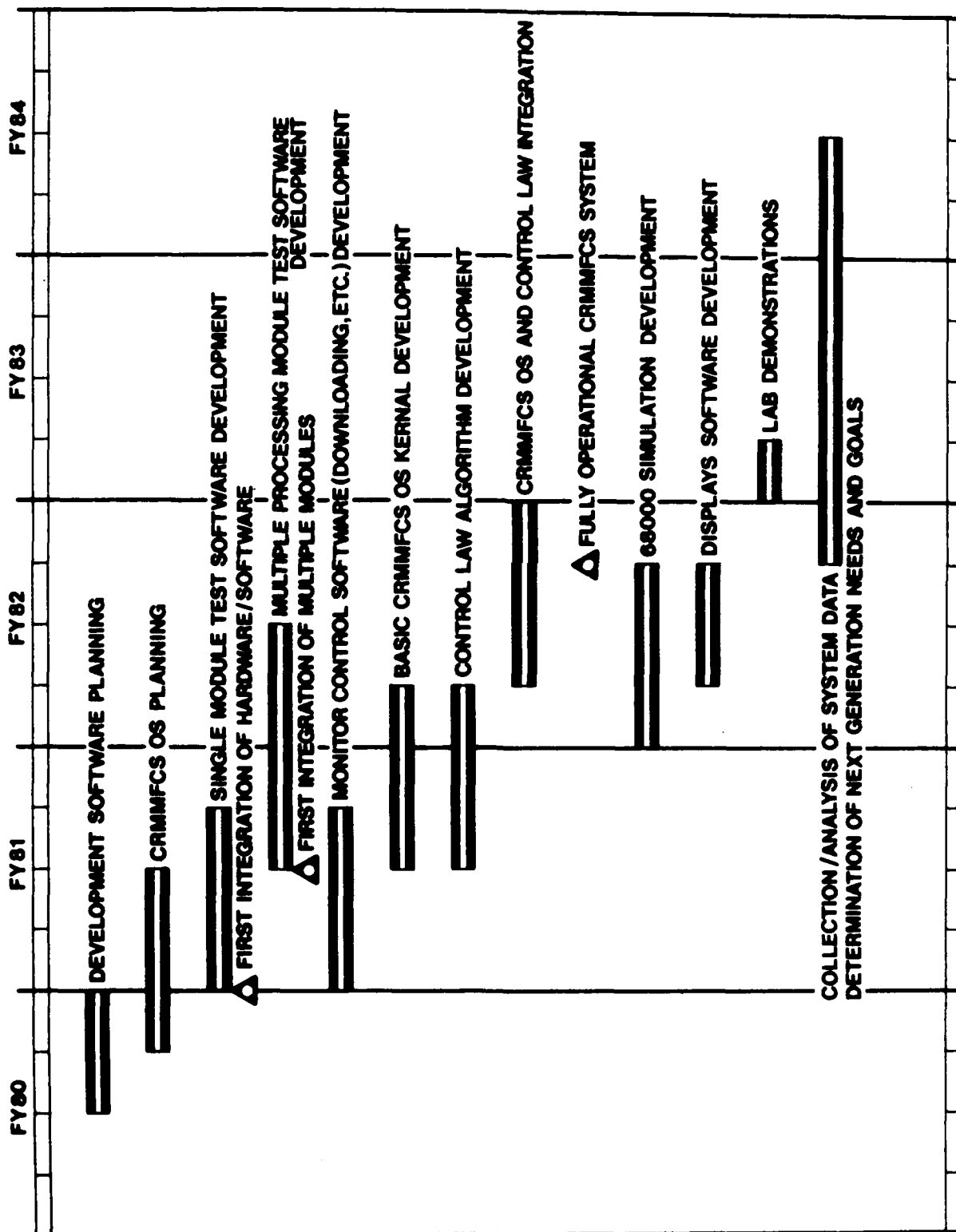


FIGURE 5. Fault Filter



## HARDWARE

FIGURE 6. Hardware Milestone Chart



# SOFTWARE AND SYSTEM INTEGRATION

FIGURE 7. Software Milestone Chart..

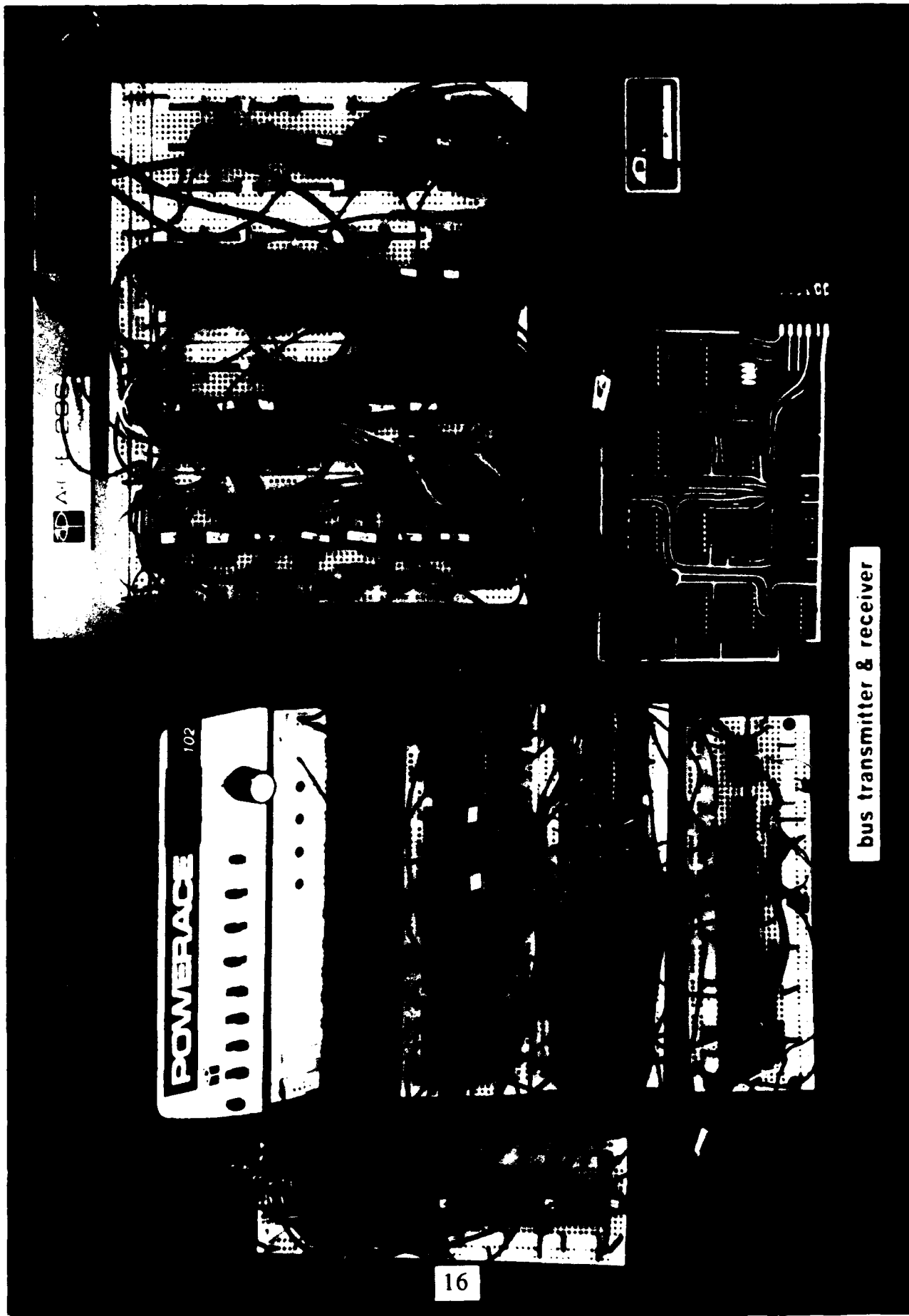
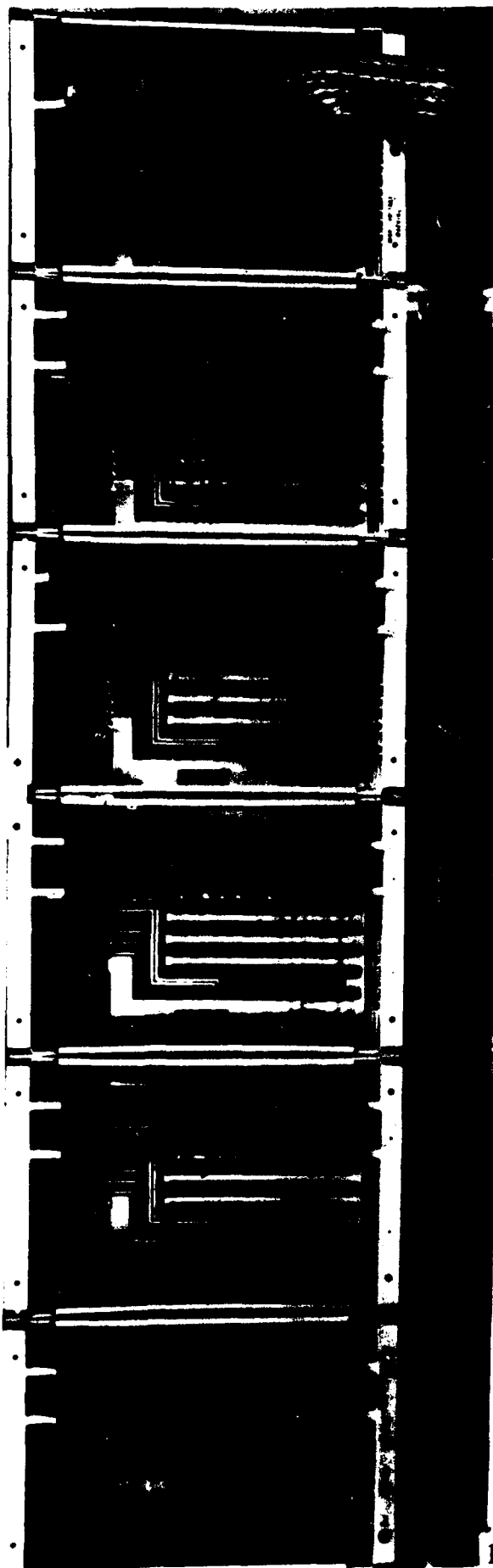


FIGURE 8. Breadboard of Receiver and Transmitter



AFWAL\FIG1



17



FIGURE 9. First Multiple Module Test Configuration

module configurations (Figure 10) for multiple processor test runs. Modifications were added to the PC boards as design errors were identified and corrected. By mid-fiscal year, a full complement of sets had been constructed. Development of the CRMMFCS OS, integrated with application control laws, had begun earlier in the period, as had the airframe simulation software and displays software work. By the end of the fiscal year, the CRMMFCS implementation was operational.

Laboratory demonstrations to industry and government personnel began early in FY83. A description of the demonstration set up is in Section 4.3. The knowledge gained from the implementation and comments received during the demonstrations were combined with the subsequent analysis work to form the basis for this report and a plan for future work (which will not be described in detail in this report). This analysis work covered the period of FY83-84.

4

#### Implementation Description

##### 4.1 Hardware

##### 4.1.1 Introduction:

As noted previously, the physical implementation of the CRMMFCS architecture began in early 1980. Communications hardware design and construction began immediately. To simplify the hardware design and development of the overall system and to keep system costs low, off-the-shelf processor boards, dynamic RAM boards, and card cages were purchased (Figures 11 and 12). The communications hardware for the multiprocessor scheme was designed and constructed in-house around these components. The constraints of integrating in-house hardware with purchased hardware was not seen as a problem. Since physical size was not an implementation consideration, small scale integrated circuit

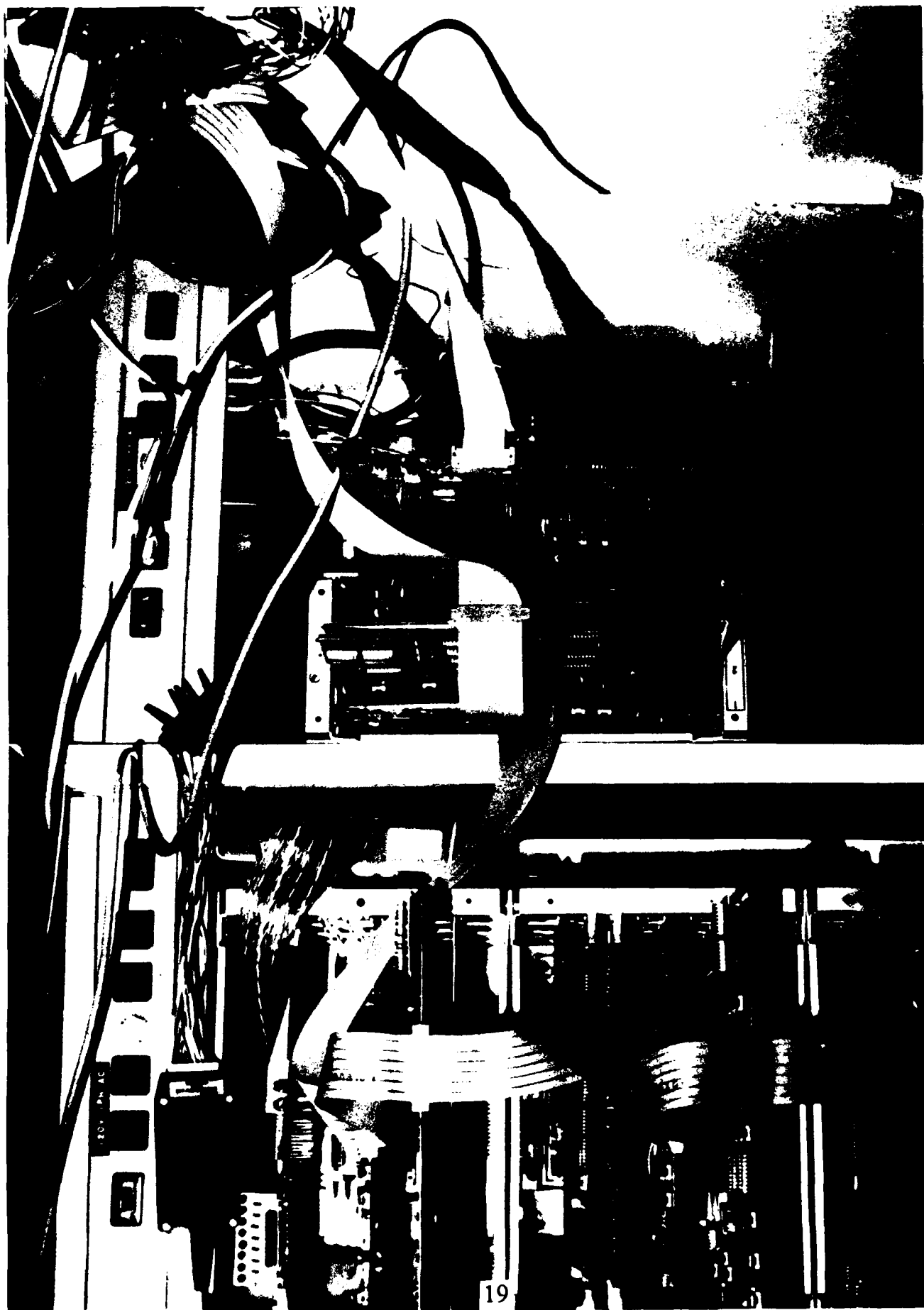
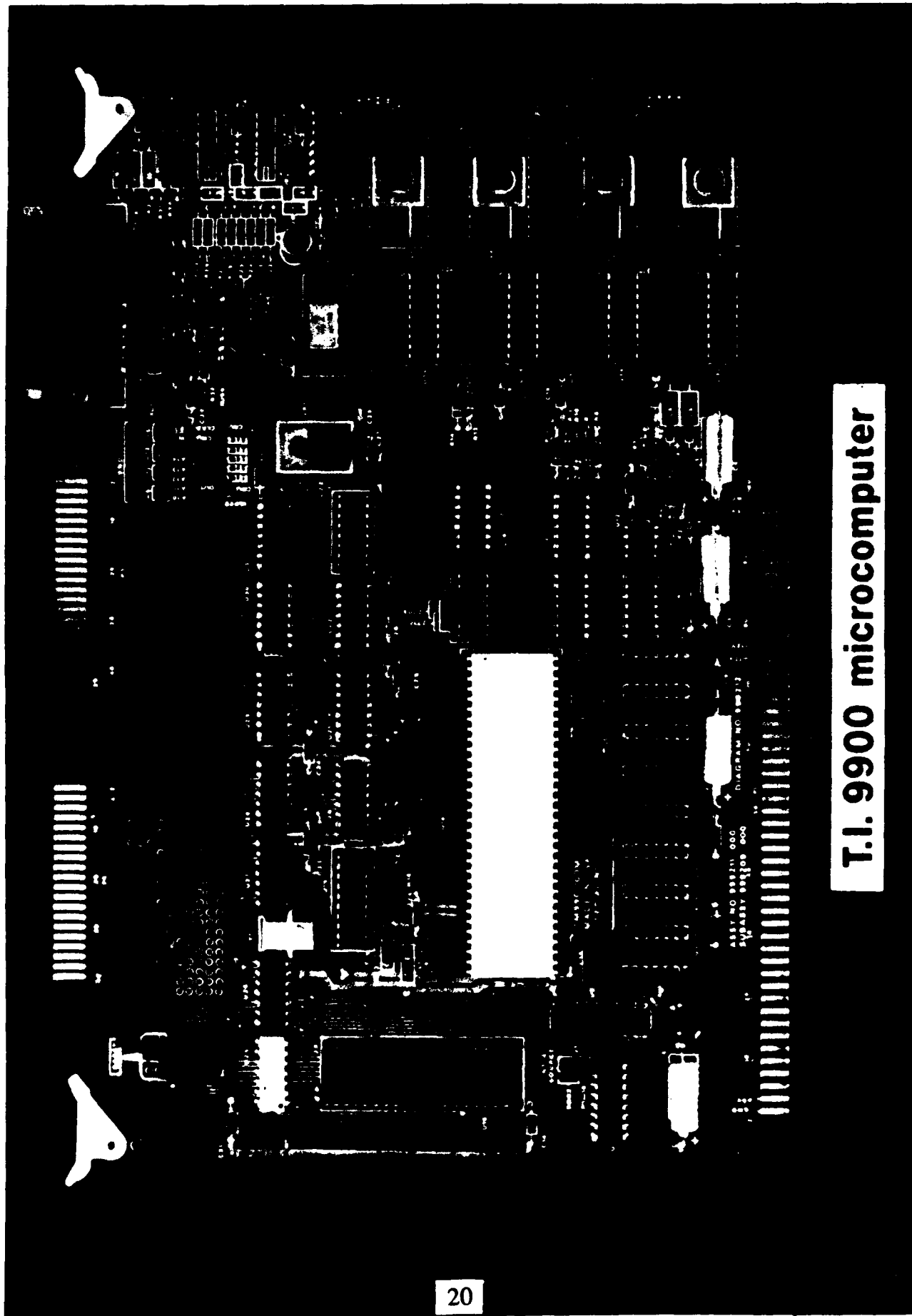
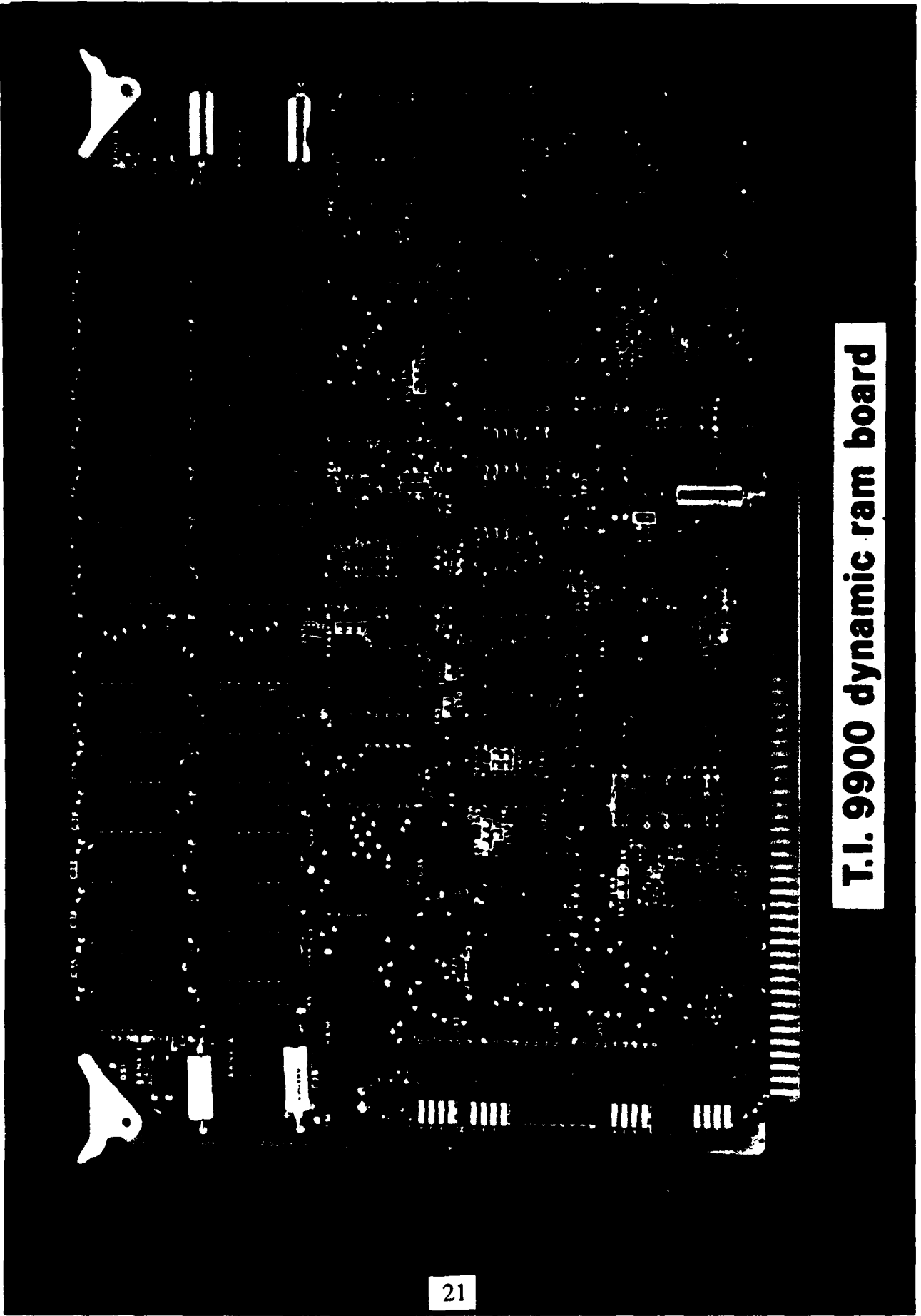


FIGURE 10. Multiple Module Test Run Configuration



**T.I. 9900 microcomputer**

FIGURE 11. TMS 9900 Microcomputer Board



**T.I. 9900 dynamic ram board**

FIGURE 12. TMS 9900 Dynamic Ram Board

chips were used exclusively in the design. This was determined to be necessary for easy modifications later. The assumption was that the reduction in size owing to the use of custom chips, programmable logic arrays, and gate arrays could be made at a later time, if desired. The emphasis for the laboratory implementation was placed on the simplicity of testing and modification.

#### 4.1.2 Microprocessor Selection:

The first implementation decision was the choice of the processor to use. Several factors were considered. The original focus of the project targeted low-cost microprocessors as the computing element. At the time of the decision, 8-bit processors were dominant; with the Zilog Z-80 and Motorola 6800 seemingly the most popular. However, 16-bit processors were beginning to appear. The first to appear commercially with any adequate support was the Texas Instruments (TI) 9900. The Motorola (MC) 68000 and Intel 8086 had been just released on the market, and development support was weak. Given the desire to keep the development and testing simple, these more advanced 16-bit processors did not seem to be good choices. The decision was left between the 9900 and one of the 8-bit processors. Upon investigating the 9900, it was determined that the on-chip 16-bit integer multiply and divide would be adequate for simple flight control algorithms. The 8-bit computations (software expandable to higher resolutions with a significant time penalty) were not seen as adequate alternatives, for even a small-scale flight control algorithm demonstration. From this, the decision to use the 9900 was made. The TI processor boards, RAM boards, and card cages were then purchased for 9900 implementation.

#### 4.1.3 Bus Structure Selection

With the processor determined, the focus changed to the design and construction of the multiprocessor communications hardware. The motivation for most of the design decisions was noted in AFWAL TR-81-3070. A brief summary of some of the design issues is included below.

When choosing a bus structure for use in a multiprocessor scheme, many factors need consideration. One such decision is whether to use serial or parallel address and data streams. Parallel busses carry the advantage of high data transfer rates. However, this advantage has the added cost of large numbers of connections and transmission lines. Fault tolerance is difficult to achieve economically. The addition of a single parity bit, for example, forces the need for another line and several more connectors. Redundant lines or busses which allow system operation with a bus failure are extremely costly. Unless the data rates provided by parallel lines are required, serial busses are preferable. Fault tolerance, however, is a major consideration in CRMMFCS. Also, since the data rates achievable on serial lines seemed adequate for the application, especially when multiple busses allow concurrent utilization, this bus type was chosen.

#### 4.1.4 Bus Access

In any kind of bussing scheme, there must be some protocol for gaining access to the communications medium. There are two basic types of busses: dedicated busses and, as we will call it in this report, open busses. Dedicated busses take one of two forms: dedicated links and dedicated transmit busses. Dedicated links are point-to-point busses between two processors, used only by one or both for communications between those processors. Dedicated transmit busses are busses where only one processor can transmit on the bus, although many may receive

from the bus. In either case, the number of busses required by the system is determined by the number of processors in the system. If more processors are added, then more busses are required.

Open busses can take many forms. In general, open busses can be accessed by potentially many processors. Bus access is gained through some bus protocol. There are two ways to accomplish this: bus contention or time multiplexing (time sharing). In contention, a scheme is used to arbitrate between processors requiring bus usage. Time multiplexing involves dividing bus access times in advance, and granting bus access whether the processor requires the bus or not.

In the previous report on CRMMFCS, justification is given to the decision to use a contention bus. In short, dedicated busses require too many lines for a system with more than a few processors; and expandability is extremely limited. In addition, in both dedicated and time multiplexed, busses are not used efficiently. Processors can control busses without any need to transmit. In contention busses, there is added overhead for bus access, however, a bus will be used only when needed. Busses will be idle if no processors have transmissions to make. With these aspects in mind, contention bussing was chosen for CRMMFCS.

There are various ways of arbitrating a contention bus. Methods of using a central bus controller were rejected because of the complexity involved in making the controller fault tolerant (i.e., a single failure point). Distributed bus control was much more desirable, if it could be implemented in a simple fashion. The method of transmission collision contention bussing used in CRMMFCS was a simple solution to implementing distributed bus control.



#### 4.1.5 CRMMFCS Bus Specifics

The implementation of the contention busses in CRMMFCS involves four logical serial busses, which any processor can transmit or receive on. Four busses are used to provide the minimal 2-fail operation condition: if two busses fail, then two are left. Two busses are the minimum to allow the bus clock/synchronization circuits to remain operational. A logical bus is composed of two busses: a serial data bus and an associated clock bus to synchronize the data. The bus pairs are dedicated to one another; when a processor gains control of a bus, it utilizes the associated clock bus. Each of the four logical clock busses operates at 1-MHz rates. This clock speed was chosen for the laboratory implementation somewhat arbitrarily; higher rates are achievable with contention bussing. Given this clocking rate and the CRMMFCS transmission width of 46 bits (see section 4.1.7), 21 transmissions can be made on one bus per millisecond; 84 per system per millisecond. Further bus specifics will be given in the discussions below.

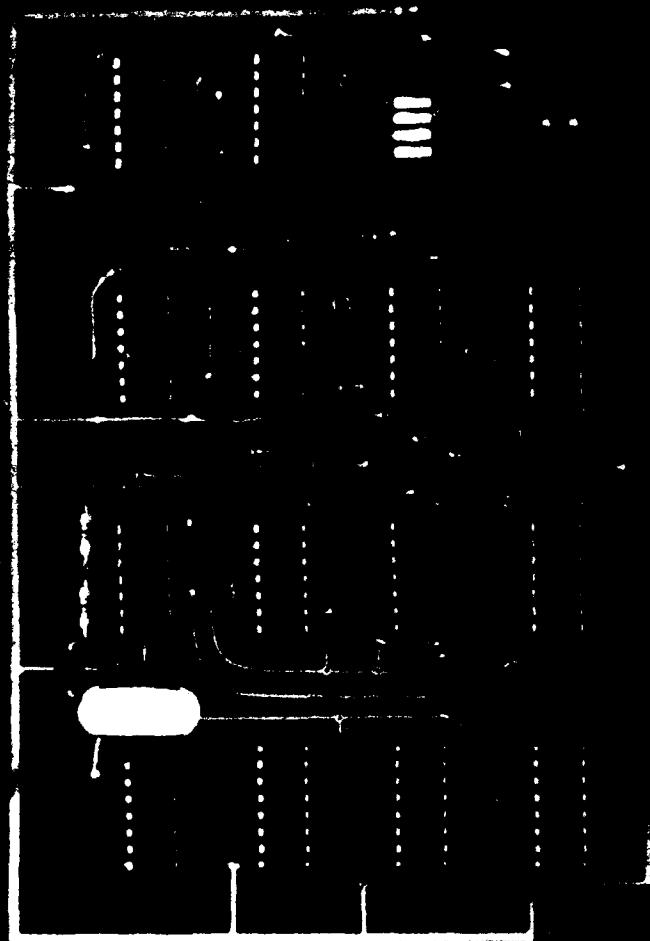
#### 4.1.6 Bus Termination and Synchronization

Given the conceptual view of CRMMFCS as described previously, the importance of processor synchronization is apparent. Since the only link between the processors is the system bus network, this is a logical choice for a synchronization reference. The clock busses were subsequently chosen to implement the synchronization. Keeping the processors synchronized at the individual clock pulse level is a complex job, and the synchronization is susceptible to bus skewing. Since the only times processors need be "synced" is at data exchanges (once per millisecond; see section 4.1.7), synchronization need only take place once per millisecond. The decision was made to place a special elongated pulse on the clock busses at the millisecond boundary. This

pulse, 8 microseconds in length, is used by the individual processors to determine when the millisecond boundaries occur. A voting scheme is applied to the clock busses to determine when a true "sync" has occurred (two or more concurrent "sync pulses").

The methods for generating these pulses and performing other bus related monitoring tasks are accomplished by a special set of circuits, which are contained in the Bus Termination circuit (see Figure 13). This hardware performs four basic functions: bus impedance termination, bus fault detection and isolation, bus clock generation, and sync pulse generation. All the functions except the bus fault monitor were implemented in the laboratory. The fault monitor, a circuit which watches both ends of a bus to ensure that data on one end matches that on the opposite end, within a certain delay tolerance, was a conceptual idea to handle the problem of bus breaks. If a break is detected, the bus is pulled low continuously to "kill" the bus. This circuit was not implemented in the lab.

Termination was a necessity for a bus long enough to connect the several processors in CRMMFCS. This was the first requirement for the Bus Termination circuit (as is evident by the name) to address. The termination of the circuit had to match that of the bus' driver/receiver chips. The DS3662 High Speed Trapezoidal Transceivers chips are intended to be used with unbalanced transmission lines terminated in 120 ohms to reduce noise. This proved to be ideal for the CRMMFCS implementation where simple ribbon cabling was used for bussing. A pull-up 180 ohm resistor to 5 volts and a 390 ohm resistor to ground was used to terminate the bus, resulting in an approximate terminating impedance of 120 ohms as specified. The results of the laboratory implementation indicated that this network worked very well, with little reflection-related variations.



**bus clock & termination**

FIGURE 13. Bus Clock and Termination Circuit

The bus clock and sync pulse generator functions work in close coordination with each other. The clock generator puts out pulses at the rate of 1 MHz. After counting 1000 pulses on a clock bus (there are four of these monitors, one per clock bus), the sync pulse generator takes control and holds the clock bus high for 8 clock pulses or until the arrival of another sync pulse. When at least two clock busses generate sync pulses, all busses are synched together. As a result, all busses are "synched" once per millisecond. As described below, each processor will monitor and "sync" on a vote of these pulses.

#### 4.1.7 Transmitter

The transmitter hardware, developed entirely in-house, is comprised of four functional areas: the interface to the microprocessor, the interface to the serial busses, the Bus Access Gate (BAG), and the sync pulse detector circuit. A picture of the final board version is given in Figure 14.

The main purpose of a separate, special purpose transmitter circuit is to free the main processor of the task of controlling the serial transmissions. In order to allow the processor to compute relatively free of the overhead burden of communications, a simple dual-port shared memory organization is used. A microprocessor in the CRMMFCS system "sees" its transmitter as a simple RAM memory buffer in which to put formatted transmission patterns. The transmission is transparent to the microprocessor.

The hardware buffer is two buffers set up in a "ping-pong" fashion (see Figure 15). While one buffer is being written to by the microprocessor, the other is being emptied onto the serial lines. At the next millisecond boundary, the pages are switched with the page last written to by the processor being emptied onto the serial lines, and the last "emptied" page being written to by the processor. A special value

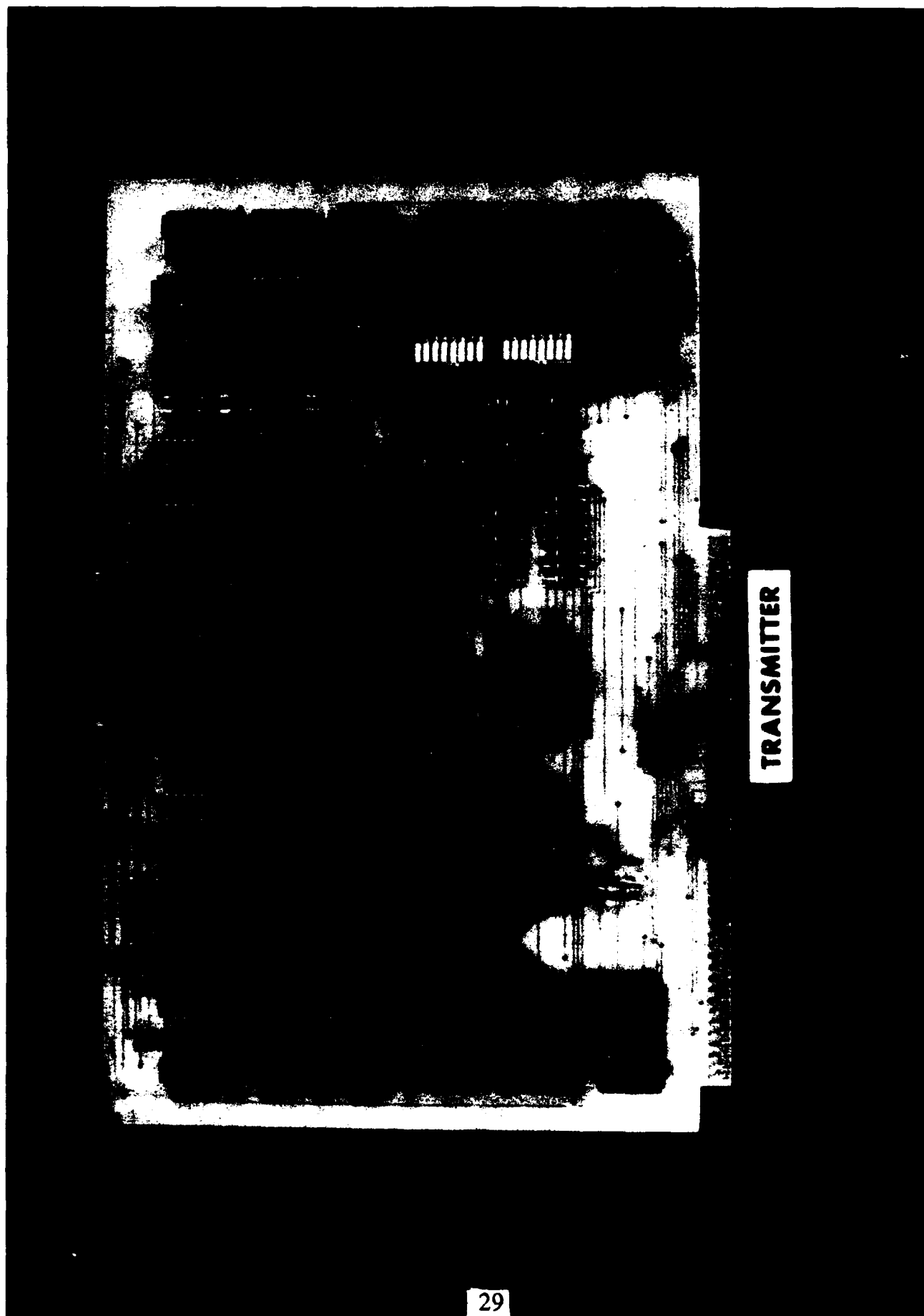


FIGURE 14. Transmitter Printed Circuit Board

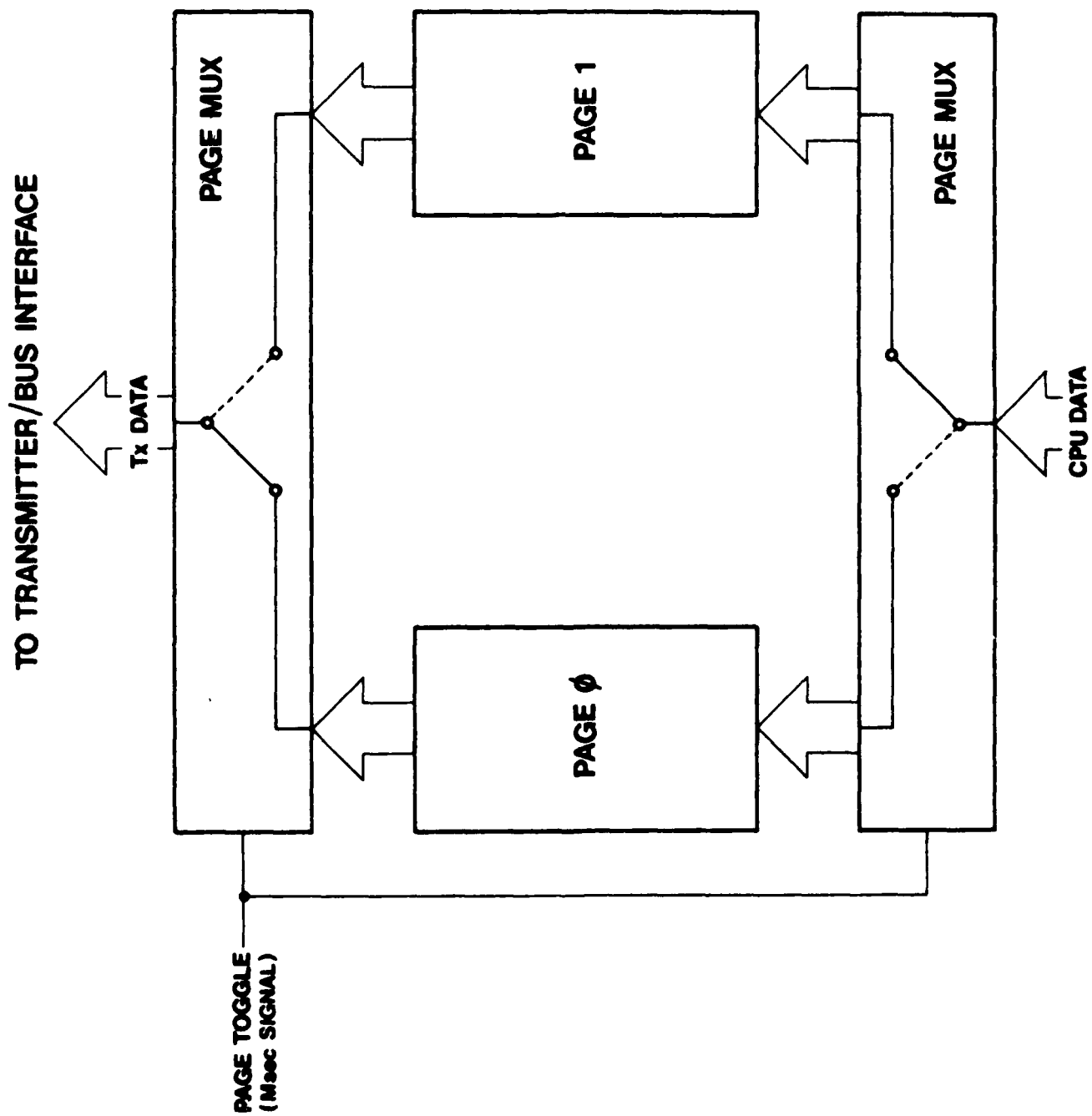


FIGURE 15. Transmitter Bus Interface

is used to mark the End-Of-Page (EOP) so that only those values placed in the buffer by the processor are sent.

The structure of the buffers are RAM pages, 256 bytes long and 8-bits wide. This allows the potential for 64 transmissions per millisecond, if the transmission lines were able to handle the load (as mentioned above, only 21 transmissions can be sent by one processor in a millisecond). Since the TI9900 is a 16-bit processor, it must perform special operations to fill the 8-bit buffer. A 16-bit transfer from the processor to the buffer is performed in two memory accesses; once normally and once with the bytes reversed. Only the low byte of the 16-bit bus is stored in the buffer. The software burden added by using the 8-bit wide buffer with the 9900 was not initially seen as significant. This burden will be addressed in Section 4.2.

At the other end of the transmitter circuitry, is the interface between the buffer pages and the busses (see Figure 16). It is the job of this section to take the data from the appropriate buffer page, select an idle bus, and output the data in a serial format with the appropriate bus protocol. The first step is to reset a counter at the millisecond boundary, to indicate the next buffer location. The next address and data word pair (which comprise the transmission) can be fetched and loaded into the shift register set.

The next step in putting out the data is to select a bus. As noted earlier, CRMMFCS uses four busses, any of which can be accessed by any processor; however, a processor can only use one bus at a time. With contention busses, if a processor fails to gain access to one bus, it moves on and tries another. This process implies an ordering in the attempted access of busses by each processor. In CRMMFCS, the priority ordering is done by bus number; in other words, bus one is the first tried, then two, and so on. This priority ordering is selectable.

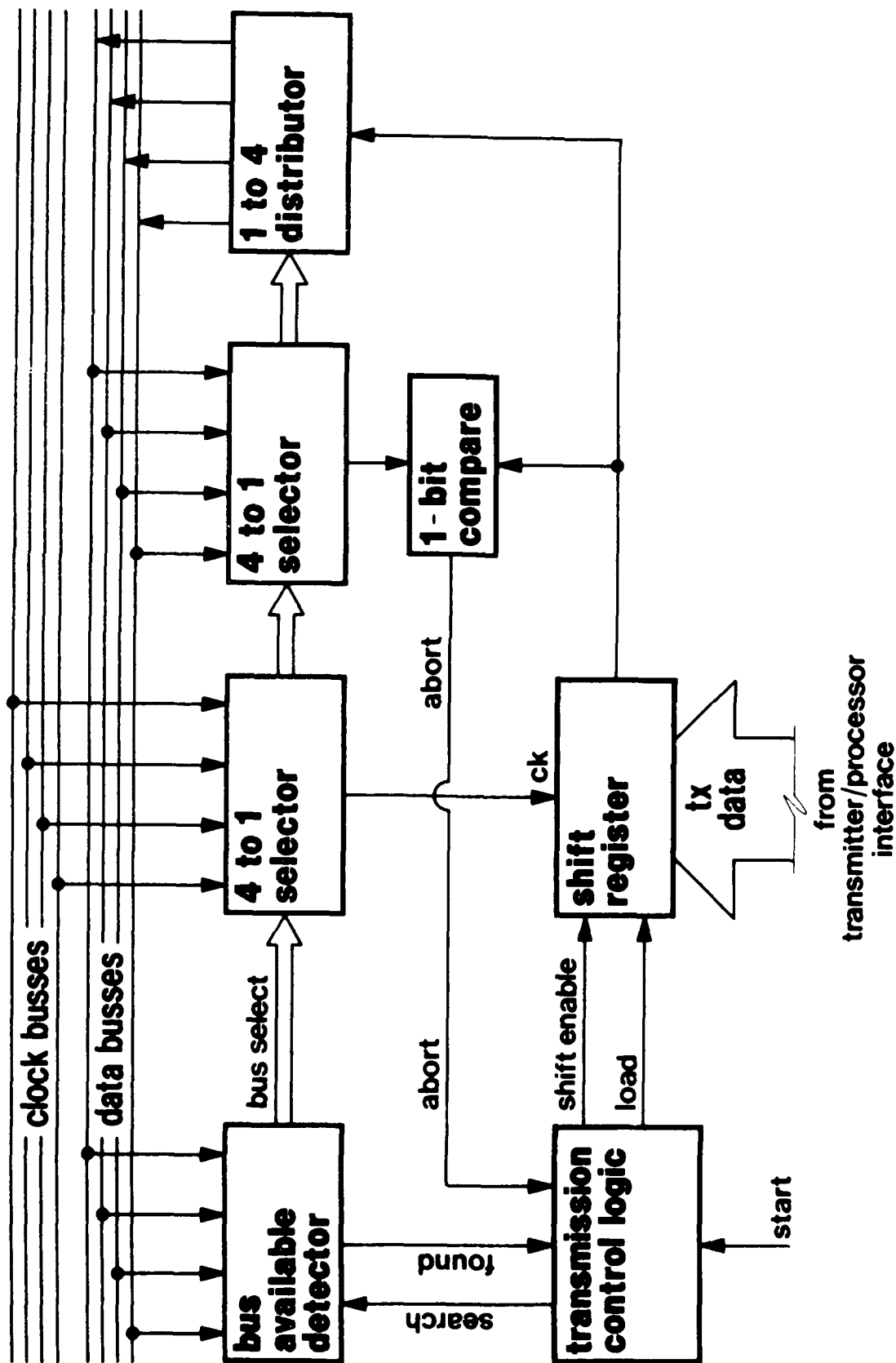


FIGURE 16. Transmitter Block Diagram



Different processors could access busses in different orders, possibly giving a more evenly distributed bus access load. This technique was not investigated during the implementation since it did not appear to be a crucial issue.

Once the bus has been chosen and the shift register loaded, the next step the transmitter takes is to contend for the bus. A contention attempt is only started if the bus has remained idle for 9 clock cycles or more. If this occurs, the shift registers begins to put the processor ID tag and address word of the transmission out onto the bus bit by bit, least significant bit first. The contention process described in the concepts Section 2 then occurs. If successful in shifting out the transmission completely, with no miscompares on the read-back, the transmitter will move the buffer counter to the next transmission. If unsuccessful, the shift register is reloaded, the next idle bus is selected, and the contention process begins anew. This process is continued until the end of the millisecond period, or until the EOP value is reached. At the end of the millisecond, the buffer pages are switched and the process restarted.

The read-back feature also plays an important part in bus error detection. A noisy or shorted bus will cause the read-back to miscompare, the same as for a failed contention. Noisy or failed busses are then skipped as a normal process. A transmission is only assumed to be correct if it is read back correctly. The two possible failure modes around this check involve long transmission delays (a processor reads back the correct data, but the transmission is damaged after a significant delay down the bus), or by a split bus. The former case will not occur if care is taken not to design the CRMMFCS bus too long. The latter case can be covered with a bus fault detector as described earlier.

When two or more processors contend for a bus, what factor in the contention scheme determines which processor wins control of the bus? The answer is a hardwired identifier tag (ID) which is unique to each processing module. This 5-bit ID tag is placed in the low order bits of the address word of the transmission, by the processor. The address word is the first portion of the transmission to reach the bus, the ID is the first bits of the address word. The ID is the contention variable for the transmission. Since each ID is unique, one processor will always win during the contention part of the transmission. This places an inherent priority for bus usage on the processors. Since the bus is effectively open-collector (see Figure 17) and inverse logic (a logical "one" is sent as a low on the bus), an ID with more "ones" in the least significant digits will have a higher probability of winning the bus. Note that different starting times of transmission attempts (within one clock pulse, because of clock skewing for example) can override this priority, though. The priority only holds when processors begin contention at precisely the same time.

Each individual transmission is comprised of a single word (16 bits) of data. The first transmission is composed of an address, telling where to put the data in the SIM, and an ID tag. At the start of the serial stream is a start bit. The 5 bit ID and the 3 least significant bits of the address are next, followed by a byte separation bit. The remainder of the transmission goes as follows: upper 8 bits of address, byte separation bit, lower 8 bits of data, byte separation bit, upper 8 bits of data, stop bit, and 9 bus idle bits. With the 9 bus idle bits included (although not actually sent by the processor; rather the bus is just allowed to float back to the high state), the total effective transmission length is 46 bits. See Figure 18 for a graphical breakdown of the transmission format.

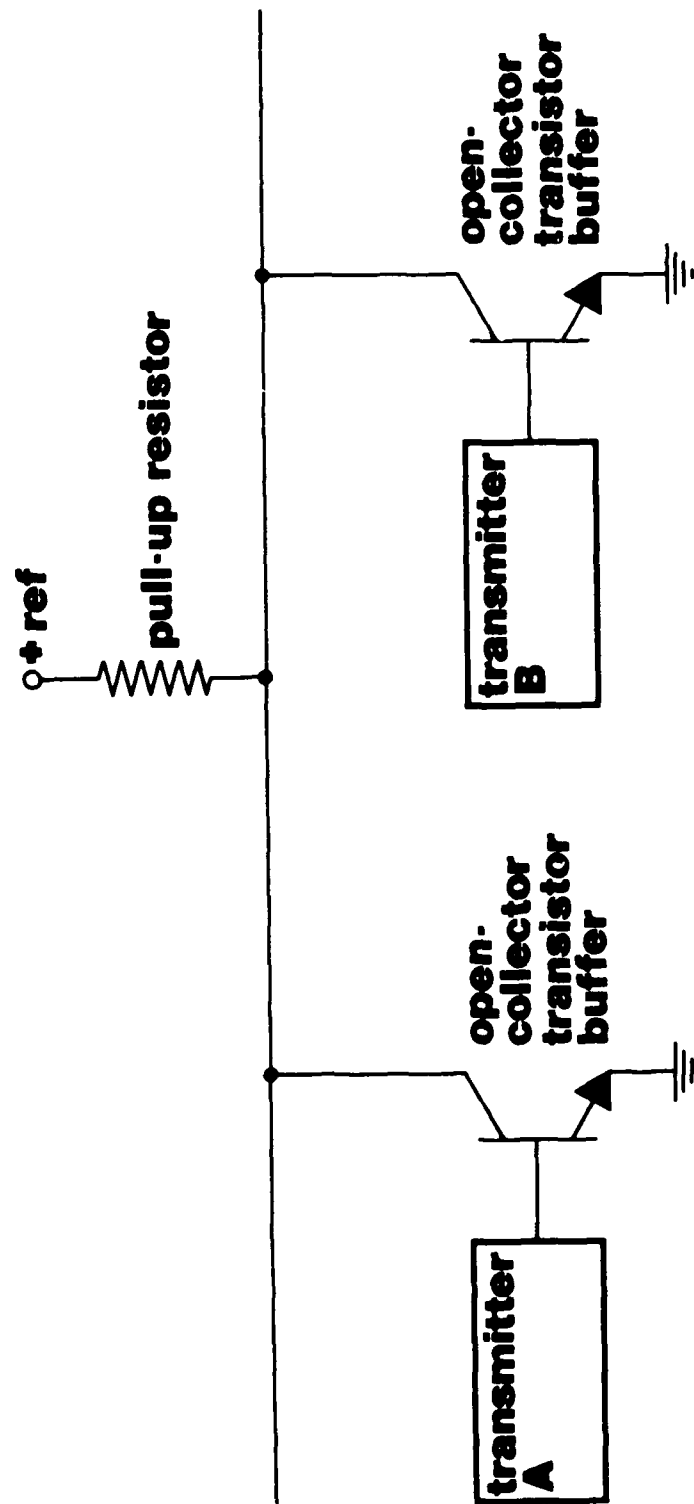


FIGURE 17. Buss Model

# transmission format

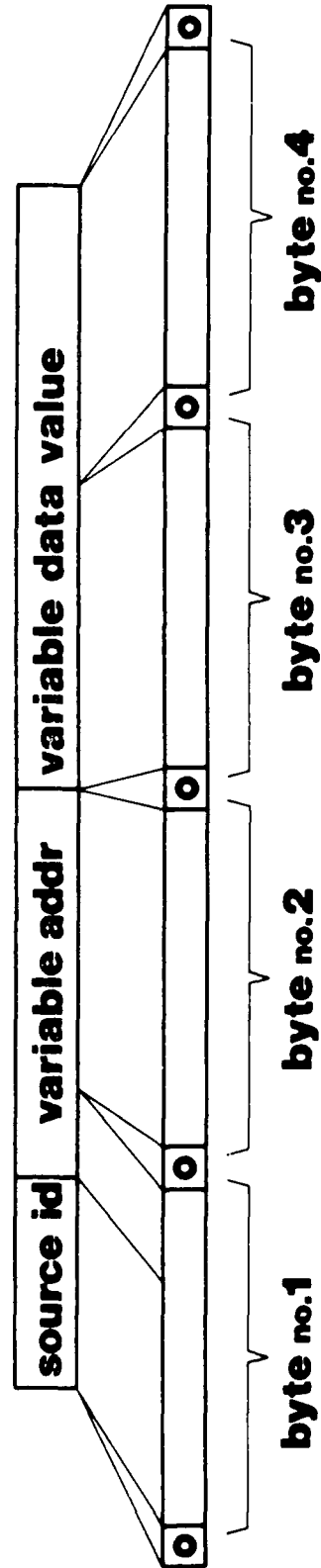


FIGURE 18. Transmission Format

The Bus Access Gate (BAG) is the only hardware feature in the transmitter that is built specifically for fault tolerance reasons. The purpose behind this circuit is to force a processor to periodically compute and deposit a key value to a port to keep transmitter privileges. The usage of this circuit from a fault tolerance viewpoint is discussed in Section 4.2. A functional block diagram is given in Figure 19. Basically, the circuit is a watchdog timer. After being reset, the circuit will count down  $n$  times. At time-out, a signal is issued which forces a miscompare in the key compare section. Therefore, a processor has  $n$  (15 in the laboratory implementation of CRMMFCS) millisecond periods to reset the circuit to avoid being timed-out. Resetting the circuit is performed simply by writing to the port. However, to prevent accidental resets, the key compare section checks to make sure that the proper key is stored. If ever a miscompare occurs (either by time-out or by a bad value being stored), the transmitter enable line is set to the negative state; disabling the transmitter.

The last of the transmitter areas is the Sync-Pulse Detector circuit. The circuit is used by the individual processing modules to determine when the millisecond boundaries occur. The circuit consists of four clock receiving units which monitor the respective clock busses for the elongated sync-pulses. Each receiving unit sends a signal to the voting section when it determines that a sync-pulse is occurring. When at least two of the four receiving units concurrently see a sync-pulse, the voting circuit signals the processor and necessary transmitter circuits that the millisecond boundary has been reached. To allow for possible skewing of the clock busses, the sync-pulse is accepted after 3 clock pulses, even though sync-pulses are actually 8 clock pulses long. Basically, this circuit is used for three purposes: page switching in the transmitter, task switching in the

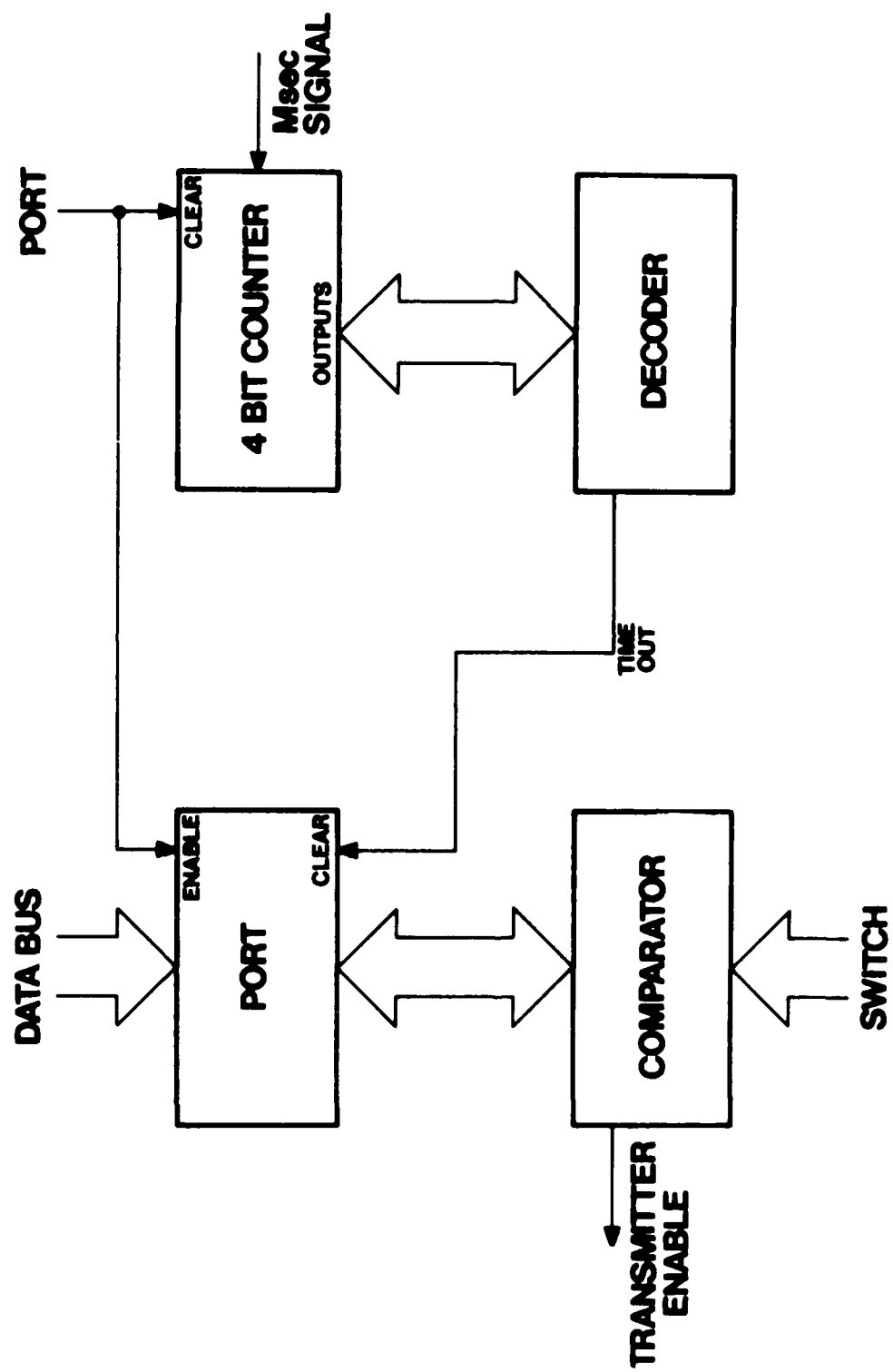


FIGURE 19. Bus Access Gate (BAG) Block Diagram

microprocessor's software (described in Section 4.2.5), and triggering the counter in the BAG. A picture of the elongated sync-pulse is given on an oscilloscope trace in Figure 20.

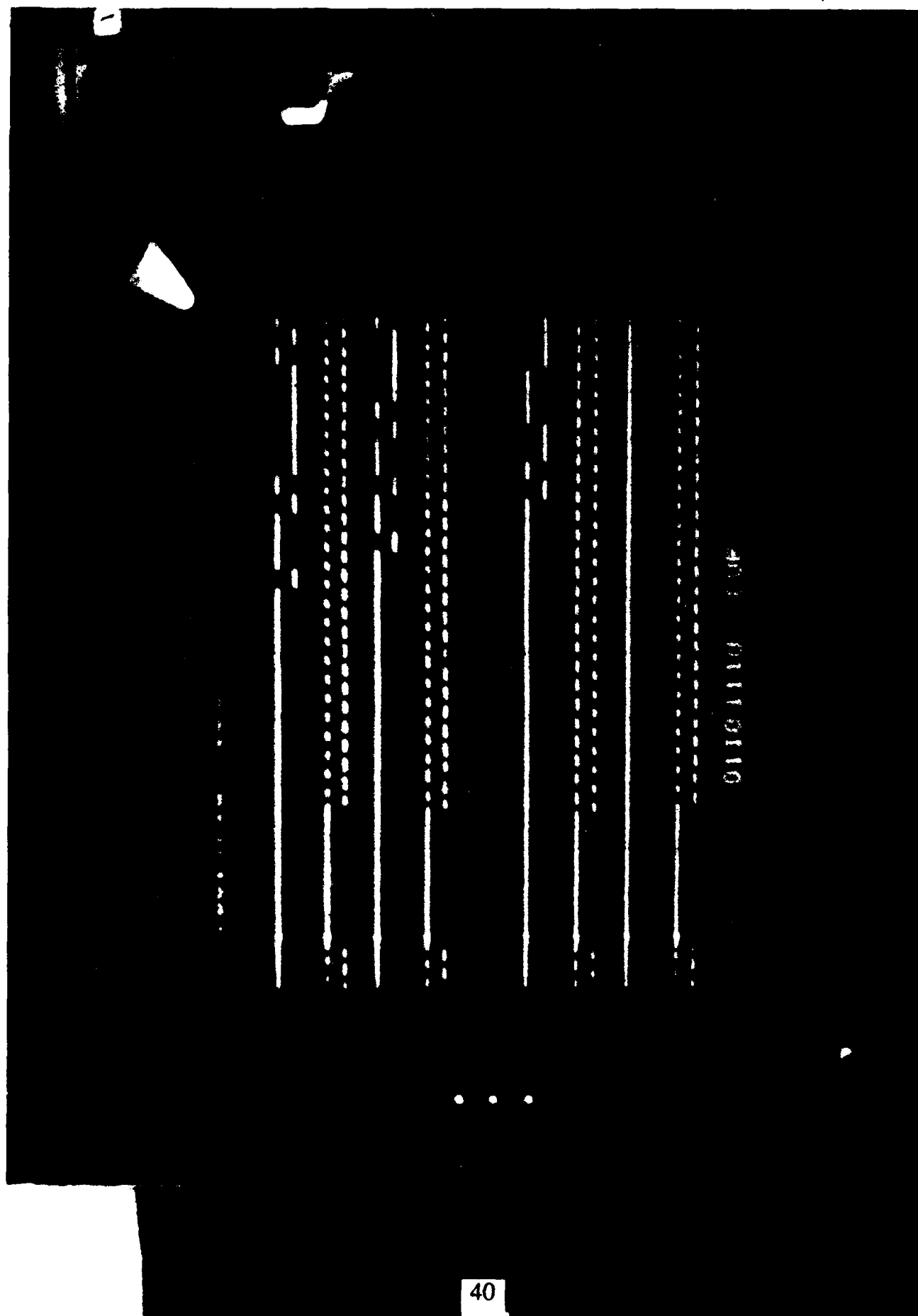
#### 4.1.8 Receiver

The receiver hardware was also developed entirely in-house. The receiver is comprised of three tightly coupled areas: the buss interface, the microprocessor interface, and the SIM. A picture of the resulting receiver board is given in Figure 21 and a functional block diagram is given in Figure 22.

The SIM, described briefly in Section 2.4, is the major part, both functionally and real-estate wise, of the receiver hardware. All transmissions received from the four busses are stored in the SIM at the addresses designated within the transmission. The microprocessor accesses these data items by reading from the SIM. The SIM is also effectively dual-port, allowing concurrent access by the bus interface and the microprocessor.

Each receivers' buss interface is made up of four receiving units, each monitors a particular data/clock bus pair. A receiving unit contains a full serial-to-parallel shift register set for receiving transmissions independently from the other units. Because of this, transmission reception on the four busses is a concurrent process. The question then becomes how the four receiving units arbitrate access to the SIM.

The problem is further complicated by the microprocessor's need to access the SIM, making a total of five units trying to access one memory on the receiver board, possibly all at the same time. This conflict is resolved in the CRMMFCS implementation by assigning priorities to the five units. The microprocessor has the highest priority and than the bus interface receiving units. Within the bus



01101110 10P

FIGURE 20. Logic Analyzer Diagram of Sync Pulses



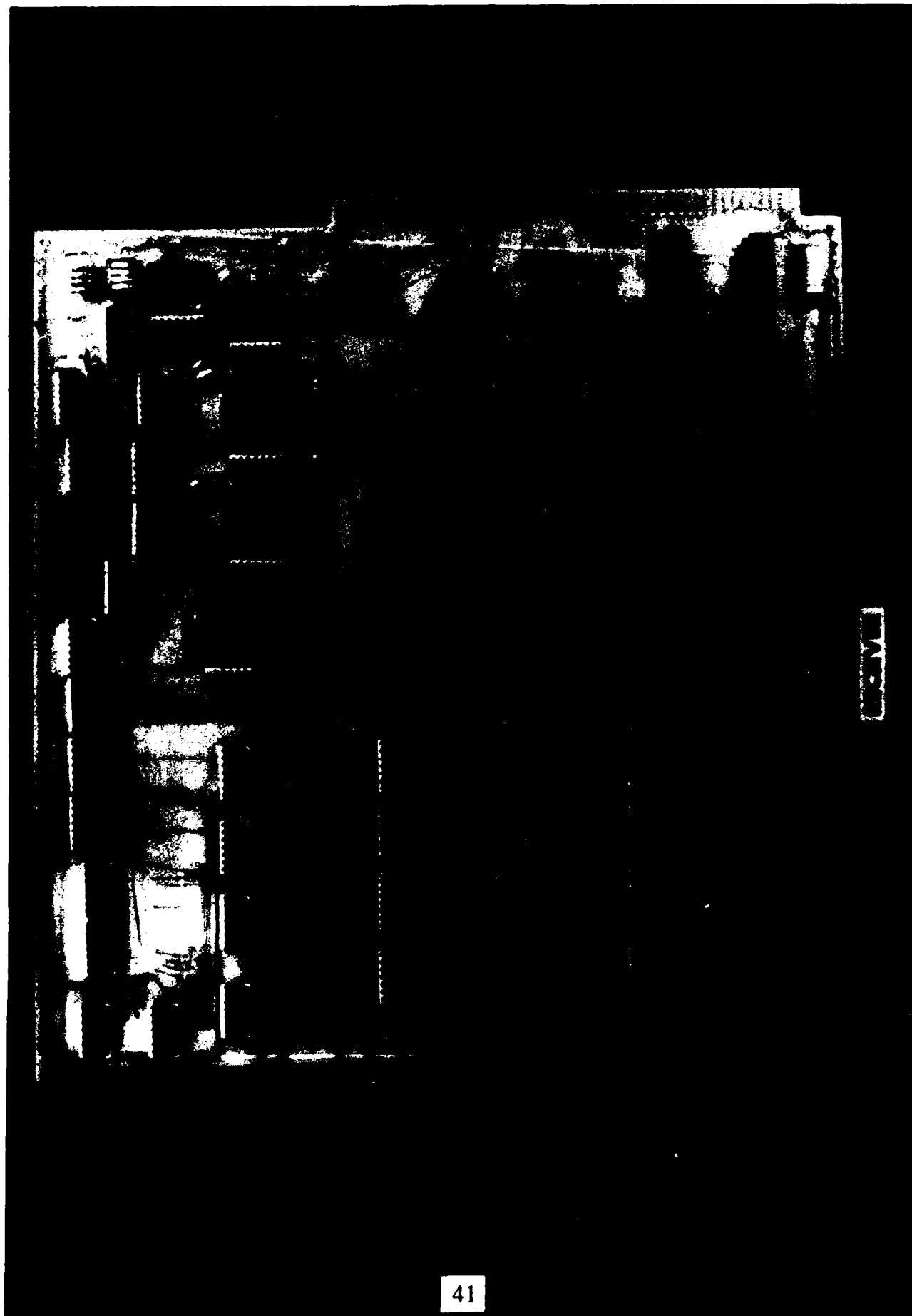


FIGURE 21. Receiver Printed Circuit Board

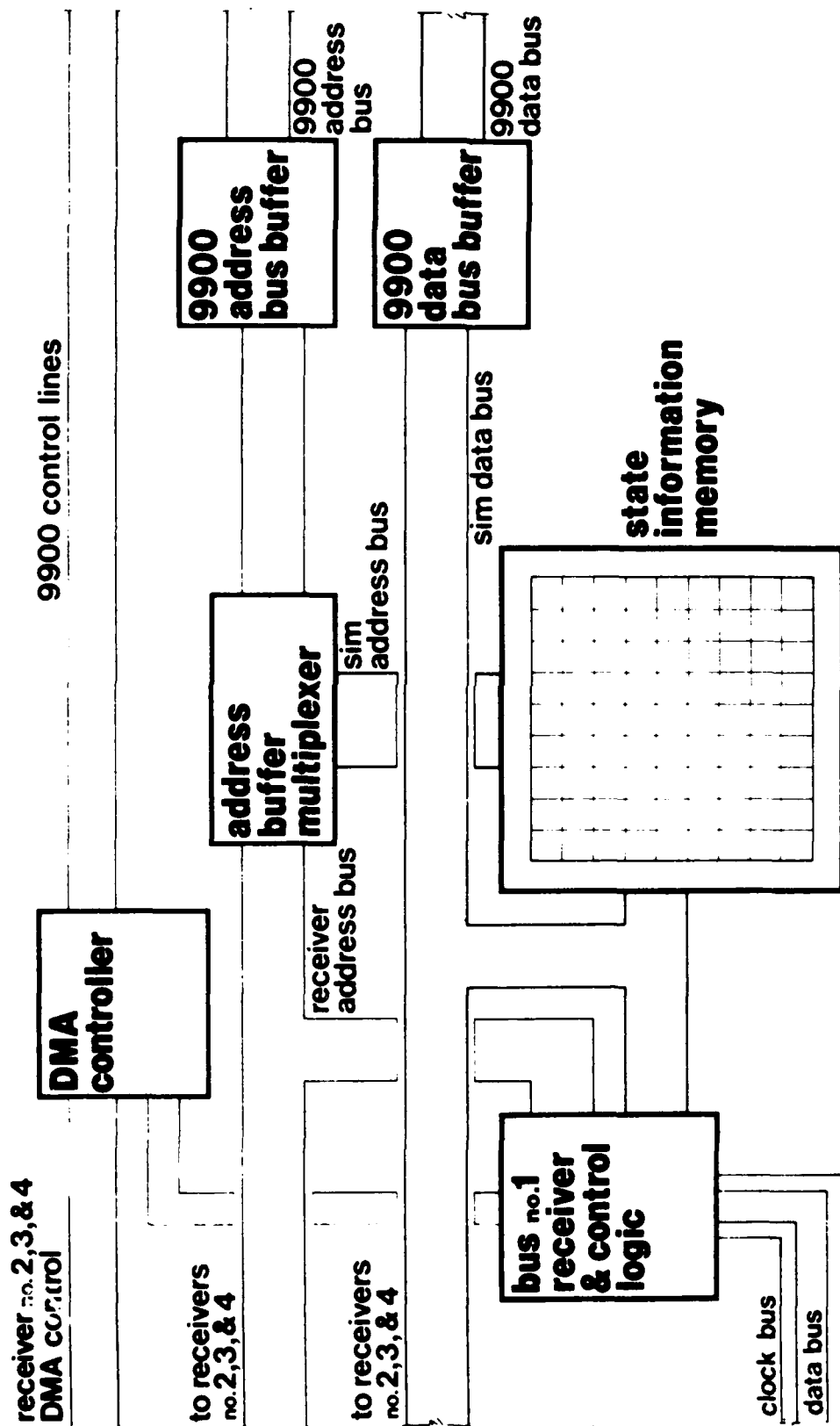


FIGURE 22. Receiver Block Diagram

interface, unit one is highest priority, then two, and so on. Double buffering is used within the receiver units to allow a transmission to be received while another is in the process of being stored in the SIM. The input shift register length is only one word, 16 information bits. The time between a data word storage in the double buffer and the successive transmission's address word, 27 microseconds, is the maximum latency time for a receiver unit to get access to the SIM. No overwriting of received transmissions will occur if this time limitation is met. As in the transmitter hardware, the priority ordering of receiving units can be modified easily. Again, since all timing requirements were met and since the ordering did not seem to be of any significance to the implementation, no further study was done on this issue.

One type of validity check performed at the receiver is the detection and proper location of the byte separation bits and or stop bit. If they fail this check, the transmission is assumed to be erroneous and is ignored. This eliminates most types and occurrences of noise on the bus since noise generally comes in bursts. Bursts of noise will very likely disturb the byte separation bits or cause a miscompare at the transmitter. In either case, the erroneous transmission will be discarded.

## 4.2 Software

### 4.2.1 Introduction

In comparison to the hardware, the CRMMFCS' software appears to be very complex. The programmer has the task of developing and scheduling code for multiple, concurrently operating processors. The entire job appears to be a nightmare of complexity. Appearances, however, are quite deceiving. Granted, the laboratory implementation of CRMMFCS was coded in assembly language, which makes the software appear

bulky and unmanageable. Also, some careful pre-application program work was needed to adequately schedule tasks for proper data transfer. However, these are not conceptual constraints of CRMMFCS. The software of CRMMFCS is, instead, rather simple, once the overall structure of the concept is understood. For example, the basic executive is 13 assembly language statements in length. Other than this routine and two short interrupt handlers, the CRMMFCS code is constructed of modular subroutines which are designed and tested separately. The CRMMFCS application design is comprised of these subroutines placed in proper order. Each processing module in CRMMFCS is identical, with the exception of the hardwired processor ID. This is also true of the software. Each hardware module is loaded with an identical copy of the entire CRMMFCS code. The hardware ID and the current state of the SIM is all that is needed to determine tasking for the system of autonomous processors. All these features lead to a operating system which is easy to build and expand from the ground up. Further details on the CRMMFCS operating system will follow; but first, some description of the development issues of the CRMMFCS implementation is given.

#### 4.2.2 Development Facilities and Initial Decisions

The TI9900 microprocessor, the processor of choice, was chosen based on its computational abilities and developmental support capabilities. To support research in microprocessor-based applications, a Tektronix 8002 microprocessor development unit was purchased (see Figure 23). This model primarily supports 8-bit microprocessors; however, full support of the 9900 was available as well as assembler support for the 68000. The strength of the 8002 is its ability to emulate microprocessor chips in-circuit (chip replacement; see Figure 24). This capability equips the 8002 with full control over the "microprocessor" execution. Test programs and the prototype hardware

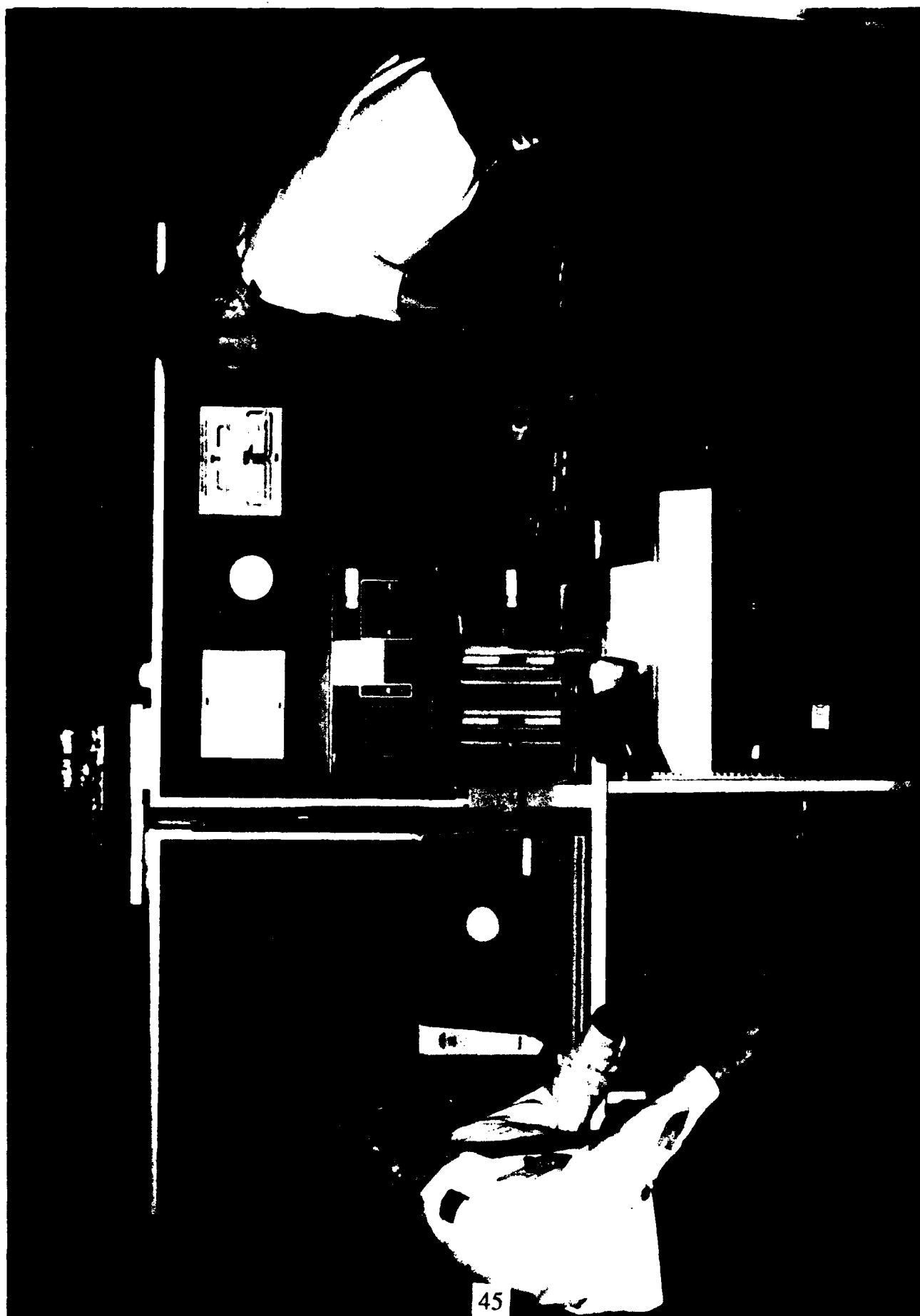


FIGURE 23. 8002 Development System

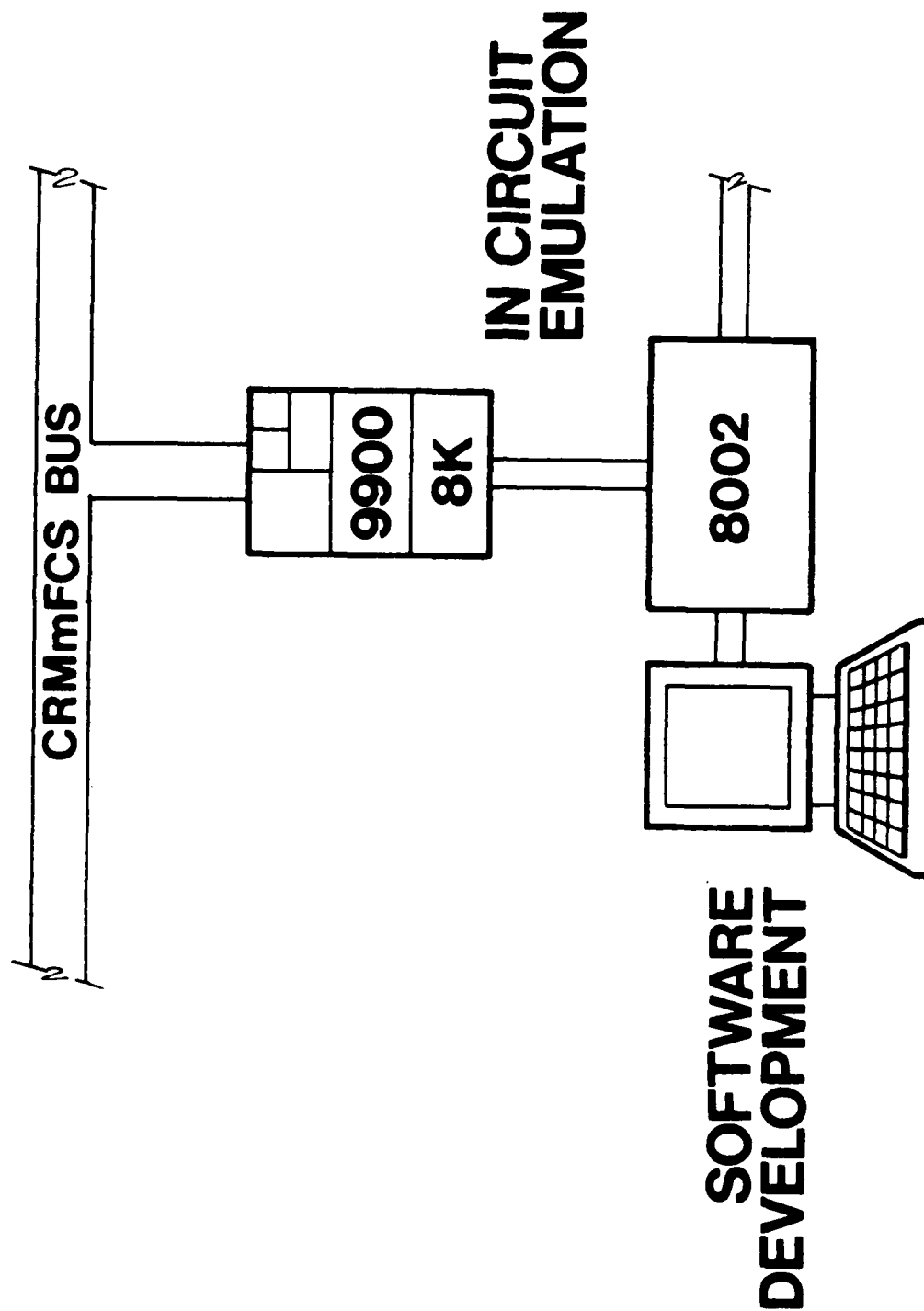


FIGURE 24. 8002 Interface to CRMmFCS Buss

are more easily debugged. The development unit contains 64 Kbytes of RAM which can be logically swapped into the memory address space of the prototype hardware, effectively substituting for nonexistent or nonoperational memory. This technique reduces the need for PROM programming during early test stages.

Several factors came into play in the decision not to use a High Order Language (HOL) to code the software. Availability was the primary variable. ADA, at the time, was not yet available for use. In general, HOL tools were not readily available for microprocessors on development systems such as the 8002. Most equivalent work using the microprocessors was done in assembly code. The introduction of the 8086/68000 generation of microprocessors changed this trend later, but again the support for these was weak at the time of the decision.

Another factor was the consideration of need for a HOL in the implementation. The 8002, with its considerable debugging capabilities, provided a very strong assembly language support environment. Given the multiprocessor-type of system, the need was for a powerful "down-to-the-hardware" type of control on the test runs. Software or hardware problems in this environment, without this low-level support, could go undetected; especially given the unique timing and synchronization procedures occurring. HOL's, of course, do provide many important advantages, particularly in code maintenance and documentation. However, hardware-level troubleshooting support for HOL's, even today, is weak; except for some specialized languages or implementations. These factors, combined with the determination that the application model to be used would be relatively small, led to the decision to program the implementation in assembly language.

Upon examining the potential effects of using HOL's in a system such as CRMMFCS, two options existed. First, the entire system

can be programmed as a whole. In other words, a single program is written for all the processors, with parallel tasks and the means of distributing the code specified. This requires a language with this parallelism capability built in. The other possibility is to have a different program for each processor. This could mean coding up several different programs. In CRMMFCS, the second option is basically taken; however, in this case, only one program is coded. This one program is loaded identically into each processor. Again, the only difference between CRMMFCS processors is the hardware processor ID. Because of this, the parallel structure of CRMMFCS does not create a problem for using HOL's. The only remaining question involves execution speed of the compiled code. This question will not be addressed in this report since the answer primarily depends upon specific compilers, and not on the CRMMFCS implementation.

#### 4.2.3 Development Support Software

One of the biggest problems in developing a parallel processing system is the testing and verification stages. Testing an individual processing unit and its associated circuitry can be accomplished in conventional ways. Once several units are connected together, the problem gets complicated. Using one development system per processing unit is far too costly, especially in systems with more than just a few processing units. As an example of one solution, consider the process of developing and testing software used in CRMMFCS.

Developing code for multiple processors can be done in a number of ways. The code could be developed on one system and transferred to ROM's for distribution to all the processing units. Reprogramming ROM's because of the detection of software errors can be time consuming during the initial code development phases. One change



means that all the ROM sets need reprogramming. This option was discarded.

A better solution for the laboratory implementation of CRMMFCS was to develop the code on one unit and download it to the other units through the communications medium. This process is accomplished easily by using CRMMFCS's shared memory concept and some simple handshaking procedures. Of course, this procedure requires some ROM-based code at each of the processing units to handle the reset and downloading processes. This code is used identically by all processors and is not changed during system development. Self-test procedures can also be built into this ROM code to check the processor out before the attempt is made to pass the software. In CRMMFCS, the ROM-based reset, self-test, and code passing routines is referred to as the Monitor Control software.

As noted above, the Monitor Control software has three main functions: reset, self-test, and data passing. Once the processor powers up, it will vector to a specified location. The vector and reset procedures are stored in ROM on the 9900 processor board. After the reset has been accomplished and initialization has been performed, a processor and communications self-test is begun. This self-test checks the the processor's internal functions, RAM, interrupt acknowledgement, and transmissions over the busses. A failure in any of these tests fails the self-test completely. Assuming all tests are executed without a problem, a status word is sent to a SIM command/status table to indicate the health of the processor. Each processor has its own table location so that a bad processor can easily be identified. A loop is entered awaiting a command to be placed in the table location instructing the processor what to do. Downloading can now take place.

Up to this point, all processors in the CRMMFCS setup perform the same functions. The command/status table has information which indicates whether a processor is off-line (not operating at all), faulty (failed in self-test; an indication of what went wrong is given), or operational. These status values are replaced with command words to initiate downloading. The processor that generates the commands is the processor emulator controlled by the 8002 development unit. This "master" processor performs the same reset and self-test procedures as the other "slave" processors. At the point of entering the command-await loop, however, the master processor jumps to a special section of code within 8002 memory so that commands can be generated.

The third function of the Monitor Control software has been described as data passing. This function actually takes a few different forms. The most obvious case is software downloading. After transmitting sections of the code to the SIM, the master processor can instruct, with a command word transmitted to the command table, the slave processors to place the code at a RAM area specified by a couple of other SIM data words sent by the master. After storing the section of code, each slave replies with a status value to the command/status table to signal completion and any error conditions. For downloads that take up more space than the SIM section can hold, the master can iteratively transmit parts of the total code.

To check the validity of the code received, the master sends a checksum at the end of the download process to another SIM location. The master then commands the slave to do a checksum over the specified RAM area. The slave computes the checksum as commanded and compares it to the checksum variable in the SIM. Again, a status reply is sent by each slave. With this, the master can determine if all

slaves received a correct copy of the software. This confirmation is crucial for a proper test run of the system.

The above process is not the only type of Monitor Control software data transfer. Consider the importance of being able to retrieve data from the slave units after a test run of the system; particularly if an error was detected. During testing, if the slave processors operate with little user monitoring capabilities, an error can occur without detection for some period of time. Externally, the cause of the error may not be apparent. However, careful examination of the faulty processor's memory could give a good indication of what happened. Without a development system attached to the processing unit (some local debugging capabilities), this is generally not possible. However, by resetting the processor and commanding it to send back a specified area of memory, the master processor and its attached development system can be used to analyze the data. Note that the self-test RAM check has been made non-destructive so that processors lost in endless loops can be brought back with a reset without destroying the memory state at the time of the problem. Basically, the process to accomplish the retrieval is the same as downloading (since this is basically an inverse operation, retrieval will be called uploading). The master processor sends the range of data it wants retrieved to the SIM, then commands the specific slave to upload. The slave retrieves the specified data, transmits it to the upload SIM section, and sends an acknowledgement to the master (through its own SIM command/status table location). The master reads the code from the SIM and stores it in 8002 memory. Again, an iterative process is used if the data to be retrieved is larger than the SIM upload area.

The last Monitor Control software function is the SIM upload. This is a slightly different version of the upload above.

Reviewing a slave's SIM state at a particular time can be as important as checking its RAM. Errors in the system can occur because one processor has a different SIM copy than another (owing to bad SIM RAM in one of the receivers, for example). However, since the SIM upload area is naturally smaller than the whole SIM, the entire copy cannot be sent at once. This creates a problem. If the upload copy is sent to the SIM, it may overlay a part of the memory we wanted to see. The SIM upload is a special case requiring that the SIM first be stored in local RAM at the slave, then transferred. The procedure for accomplishing this is similar to the upload procedure above.

The above master-generated commands achieve some flexibility in usage when the command file capability of the 8002 is used in coordination with breakpoints. Each command that the master can send has a different routine associated with it in 8002 memory. At the end of the routines are jumps to a specific location which triggers a breakpoint. This location is the same as jumped to by the master when the command-await loop is left. In this way, command files can be set up to control the initialization and execution of the command routines. After the breakpoint is hit upon completion of the reset and self-test procedures in the master, control is returned to the 8002. The user can order command file sequences to download, execute, stop, retrieve, and so on.

Note that CRMMFCS is not a master/slave system, it's an autonomous system where control is distributed. Any system processor can do any system task; all are equal. The master/slave procedure in the Monitor Control software was set up to take advantage of the 8002's capabilities as much as possible. In fact, to extend the usage of the capability further, often the download master processor is not used in the CRMMFCS test run. Instead, this processing module is used as a

monitoring source after the download and execute procedure is finished and the test run has begun. While the CRMMFCS processors are executing, the user can use the 8002 to examine the SIM, either under manual or program control.

The case of the SIM Monitor is the most significant. The previous technical report on CRMMFCS defined the SIM Monitor in terms of usage on a TRS-80 microcomputer (attached via a parallel port to a 9900 processor). A closer examination identifies an enhanced capability that exists utilizing the 8002 through the master processor. Since the emulated processor is the center of the CRMMFCS test run initialization, the 8002 was the most logical choice for a SIM Monitor. After some investigation, the process of transferring acquired data to 8002 disks in a suitable format was determined. After storage, various tasks can be performed; ranging from simple dumps and printouts to transfers to other systems for plots to be made. The SIM Monitor functions can also be applied to data retrieved through uploads from other processors. In this way, all test runs of CRMMFCS could be set up, controlled, and analyzed from one system. The Monitor Control and SIM Monitor software proved to be crucial to the successful completion of the CRMMFCS implementation.

#### 4.2.4 Typical Multiprocessor Software Problems

Before proceeding into the description of the operating system (OS) of CRMMFCS, a short discussion of the typical problems of the multiprocessor programming environment is appropriate. This topic covers an extremely large subject area, and most of the solutions reached are application and architecture dependant. We attempted to give a brief and general sweep of the problems associated with multiprocessor software. This discussion lends some insight into the CRMMFCS OS design rationale.

The three main areas of concern are: sharing and protecting resources such as memories, tasking, and handling messages and data. Resource management is important to insure that the proper processors use the proper resource at the proper time. For example, if a processor attempts to use an array in a shared memory when another is writing to it, the first processor may not get all values from the same time frame. Correct tasking requires many things; examples include: guaranteeing all processors are being utilized efficiently, making sure that only one processor executes each individual task, and making sure that all tasks get done within the required time. Message and data handling involves the aspects of making sure transmissions from one processor reaches its intended destination, making sure that the data needed by a processor for its intended task is readily accessible, and making sure that data latency and other time dependencies are met.

Various solutions exist for each of the categories above; most are system and application dependant. Data synchronization and protection is often accomplished with software constructs such as semaphores or monitors, or with hardware interlock mechanisms. Data flow represents another type of solution. Tasking tends to be dynamic, involving the physical passing of software; or static, where tasks are assigned permanently. Message and data passing often is accomplished through central shared memories, strict point-to-point message passing, or broadcast transfers. Of course, these are not the only solutions available.

CRMMFCS handles each of these three problem areas by simple means, made available by its intended application and unique architecture. As will be explained further, CRMMFCS uses predetermined task ordering. Data transfer is naturally synchronized, no conflict of data usage will occur as long as the user sets up his tasks according to

the constraints of CRMMFCS (data transfer latency, required overhead, etc). Data validation is provided through the use of triply redundant variables in the SIM and triplex tasks. Tasking is somewhere between the dynamic and static definitions given above. Tasks are predetermined and preordered, but not permanently assigned to individual processors. Instead, tasks are dynamically reassigned at periodic intervals. Message and data passing is accomplished through the SIM, or Virtual Common Memory (VCM) as it is also referred to. This is a shared memory, but only logically. Each processor has its own distributed copy. Transfers are performed by broadcasts of individual variables.

#### 4.2.5 The CRMMFCS Operating System

##### 4.2.5.1 Control Requirements

To achieve a better appreciation for the CRMMFCS design, consider the application the system is intended: flight control. Consider for now the more general area of control. In these systems, the operation is somewhat predetermined and the response deterministic. A specified output will occur given a current control state and inputs.

Functions in this environment are computed at regular intervals. The timing of the input sampling and the computation of outputs is critical to the stability of the system. Because of this, the operation of the system is made to be iterative; repeating computations at specified frequencies. Overhead required by the system must be built around these frequencies to insure proper system response.

Note that these requirements should not constrain the system to one function computed, or one frequency. Different modes of operation (owing to different external conditions or jobs) may require a change in the function being computed, and this function may require a different frequency of computation. The case might also be that two or more functions at different frequencies be concurrently computed. These

special cases do not really add much to the system requirements, except that the means of ordering and selecting tasks must be flexible. The main point is that the tasks are predetermined and the computational requirements must be enforced stringently.

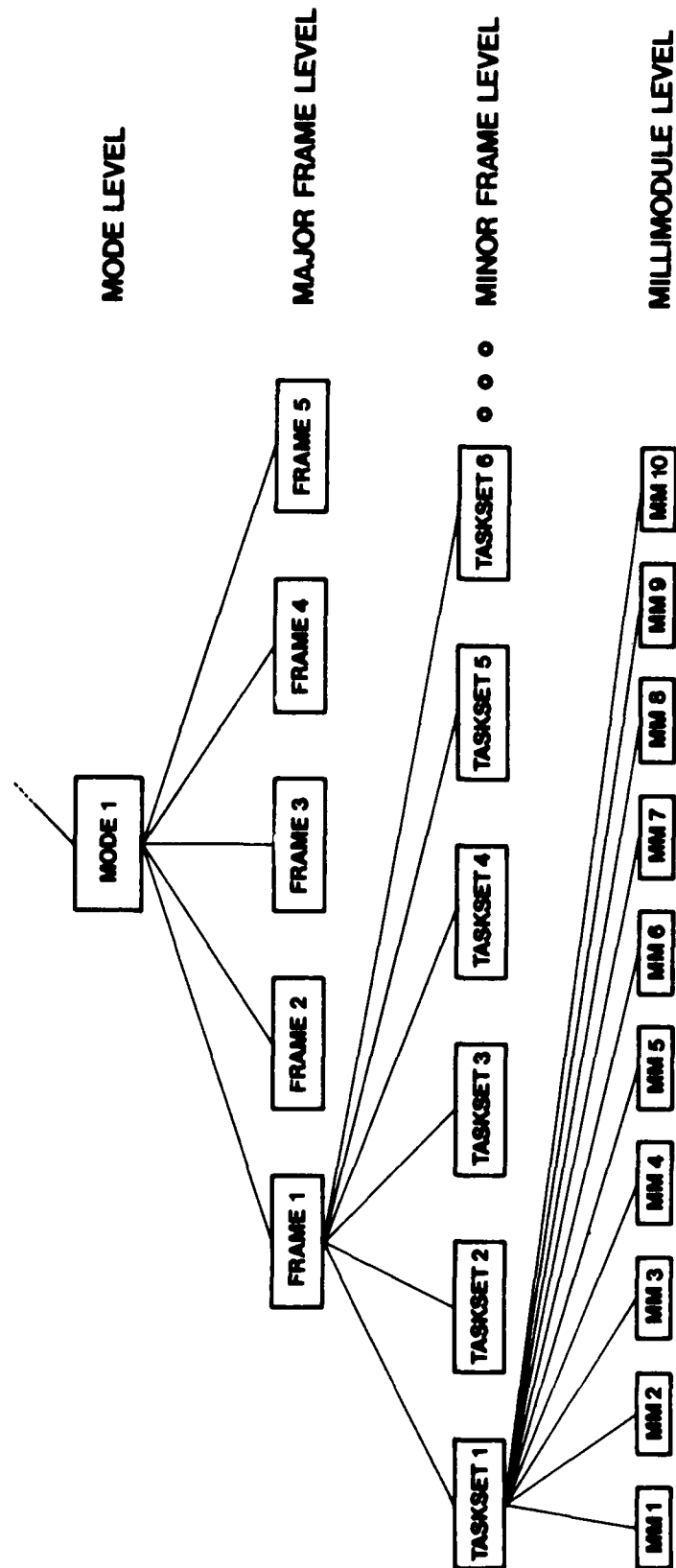
#### 4.2.5.2 Modular Structure

CRMMFCS accomplishes the above requirements with a rather simplistic and modular software structure. Remember each processing unit has its own copy of the software and data, software needn't be transferred from one processor to another. Tasks are made up of a number of modular subtasks. These tasks are combined to form the necessary functions at specified frequencies. CRMMFCS uses a hierarchical (tree; see Figure 25) task list structure to organize task and subtask orderings. Different modes of operation are entered through the different branches of the upper tree structure. Each mode branch is made up of  $m$  task lists (frames) in sequence, where  $m$  is determined by the frequency of computation. A task list is a set of  $n$  tasks for the  $n$  number of processors to perform. Each task (or task set) is composed of a set number of modular subtasks (millimodules); each designed and tested individually, separate from the task ordering process. The placement of these subtasks is generally flexible since the execution time length of each is common. This procedure of constructing the system of common length subtasks into this modular and hierarchical structure has been termed "quantized" software.

#### 4.2.5.3 OS Kernel

In order to access and execute the subtasks, as determined by the task assignment tree, an executive function, or kernel as it will be referred to in this report, is needed. The CRMMFCS kernel is short (13 lines of assembly language code), and its flowchart is given in Figure 26. At the beginning of a new frame (a new list of tasks), the





## TASK TREE

FIGURE 25. Task Assignment Tree

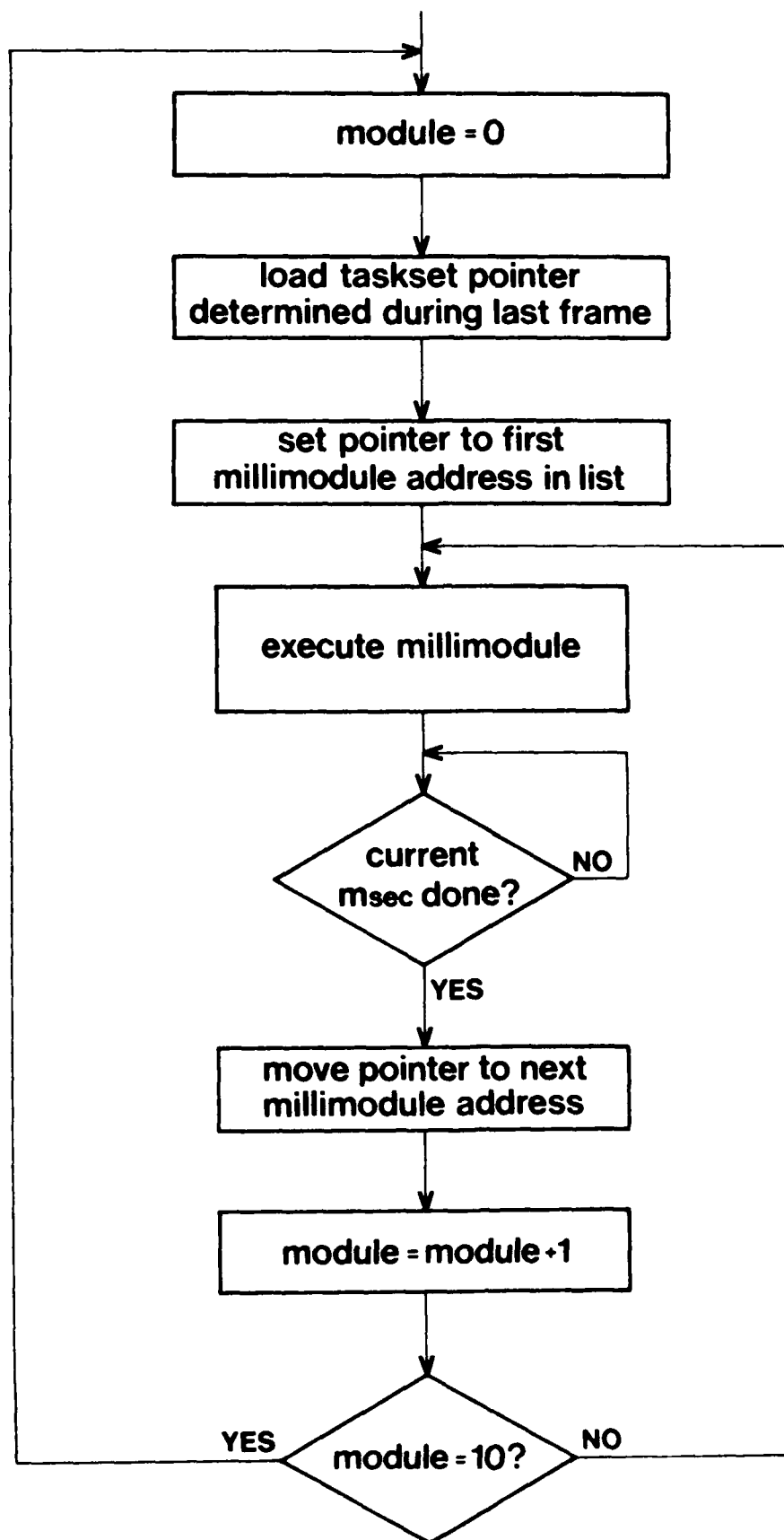


FIGURE 26. Operating System Kernel Flowchart

kernel acquires the task set address from a variable loaded from the last frame. The kernel fetches each subtask address from the list, one at a time, and executes the module. Upon return from the subtask, the kernel awaits the end of the current millisecond period, then increments the subtask counter and proceeds to the next subtask. The procedure repeats until the end of the current task set. At that time, the next task set address is fetched from the variable and the process is repeated. Note that the new task set address is computed in an overhead subtask scheduled into task lists. The kernel merely uses the task set address pointer to find subtasks to execute, and maintains subtask synchronization.

#### 4.2.5.4 Millisecond Interrupt and Subtask

##### Synchronization

The only time a CRMMFCS' processor should be interrupted is when the sync pulse interrupt signal occurs, described earlier in Section 4.1. The interrupt should only occur when the processor is in the kernel. The reason is that all subtasks are of uniform maximum length. To simplify code design, the actual subtask execution time was allowed to be less than the common time, one millisecond. When the subtask is finished, the processor returns control to the kernel, to await the end of the millisecond period. To achieve efficiency, one would like each subtask, or millimodule as they are called, to use up as much of the allotted time as possible. The programmer should not be constrained to designing code of precise time lengths. The interrupt signals the end of the allotted time. If software, other than the kernel, is interrupted, an error condition occurs taking the processor offline, on the premise that it is no longer processing code correctly.

#### 4.2.5.5 Communications Hardware From the Software Prospective

The interaction of software and communications hardware has been briefly addressed in the Hardware section above. The software needs only to perform memory reads or writes to accomplish data transfers. The software interacts with the receiver via reads from the SIM. The software really never "sees" data coming into the processing module. Reads are coordinated with data arrivals through proper task scheduling.

SIM variables are assigned specific addresses within the CRMMFCS memory map. Each transmission is composed of two words: an address word, made-up of address and ID information, and a data word. Both words are stored in the SIM. A variable is more than just the data that it represents; it contains information identifying the originator of the data and some indication as to the validity of the information. The address and data words are stored sequentially in SIM memory with the address words on the even word boundaries (address divisible by four bytes) and the data words on the odd word boundaries. Since the SIM is 8 Kbytes in this implementation, only 11 bits are required to specify SIM variable addresses in transmissions (13 bits are required for 8 Kbytes, but 4 bytes make up one variable). In the implementation memory map, the SIM resides from \$2000 to \$3FFF. Note that the SIM is read-only for a processor-to-memory access. Writing to the SIM always requires transmissions.

The transmitter also appears as a segment of memory to the software. Although the Hardware section may have left the impression that the transmitter buffer is FIFO, this is not the case. The transmitter buffer (the page as seen at that time by the processor) is a block of memory addressed from \$F400 to \$F4FF in the CRMMFCS memory map. Transmissions can be ordered arbitrarily in the buffer; however, the transmitter hardware will empty the buffer beginning at \$F400,

moving toward the end of the buffer. Since proper planning can ensure that all transmissions will be sent, ordering is not an issue. Therefore, programming fills the buffer from the beginning. The EOF flag (a zero in the ID field of an address word) signals the end of the buffer to the hardware.

Formatting of transmissions must be done in software. Data words require no formatting, but address words do (see Figure 27). To accomplish this, the address of the variable (the address word's address) is put into a processor register, the least significant two bits are zeroed out, the address is shifted left three times to put the eleven address bits into the most significant positions, and the processor ID is OR'ed into the least significant five bit positions. This creates the 16 bit address word, which is stored before the data word in the transmitter buffer. The entire transmission is stored in the buffer a byte at a time, since the buffer is only 8 bits wide. The least significant byte of each word is stored first, then the most significant. An example of 9900 assembly language code which formats and stores one transmission is given in Figure 28.

#### 4.2.5.6 Tasking and Reconfiguration

Up to this point, the discussion of tasking in CRMMFCS has been limited to the individual processor. What remains is how multiple CRMMFCS processing units work together and perform reconfiguration. A clear definition of the term task will prove to be beneficial in the discussion to follow. In formal jargon, a task is the time spent by a processor volunteering and executing its selected job sequentially until completion. In other words, a task is the code executed by a processor between reconfiguration times. Since a task is made up of a set number of subtasks, and each subtask is of uniform length, a task is of a

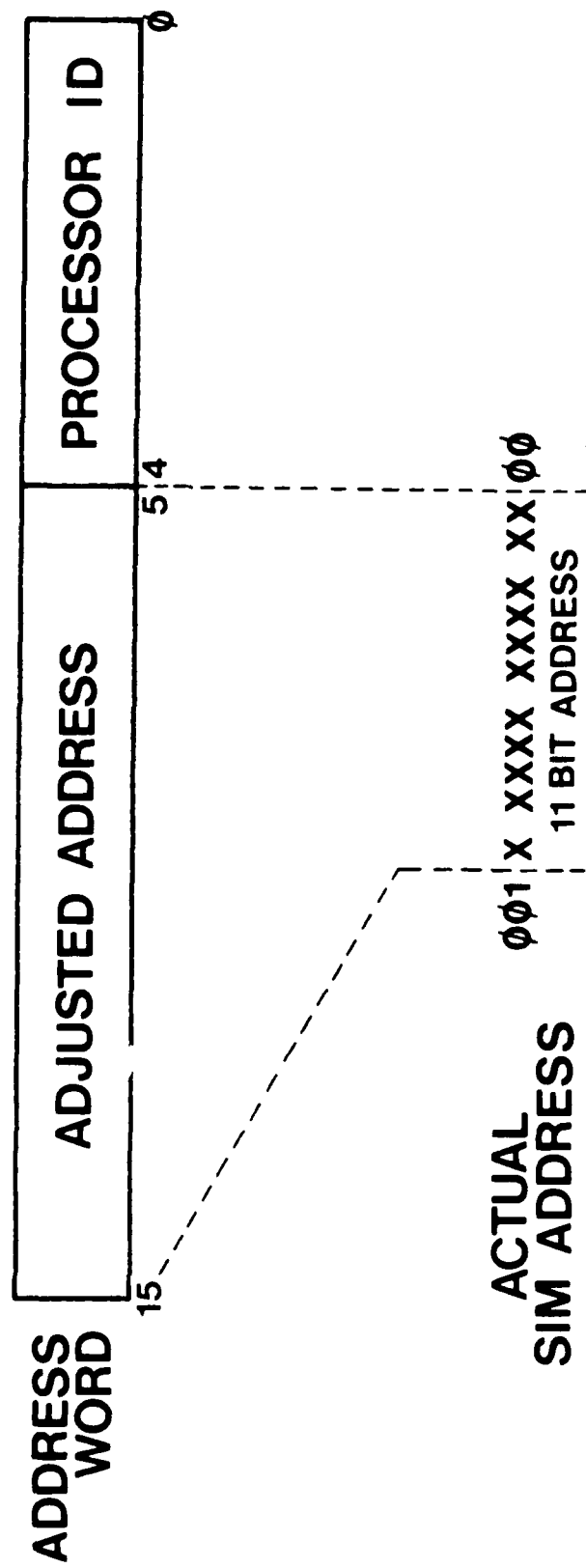


FIGURE 27. Address Word Formatting

```

      ○
      ○
      ○
LI    R1, SIM ADDRESS.....; GET SIM ADDRESS
SLA   R1, 3.....; SHIFT INTO HIGH 11 BITS
MOV   PROC.ID, R2.....; ADD IN 5 BIT PROCESSOR ID
ANDI  R2, 001EH
A     R1, R2

MOV   R2, TRANSBUF.....; MOVE ADDRESS WORD
SWPB  R2.....; TO TRANSMITTER BUFFER
MOV   R2, TRANSBUF+2
MOV   VARIABLE, TRANSBUF+4.....; MOVE DATA WORD TO
SWPB  VARIABLE.....; TRANSMITTER BUFFER
MOV   VARIABLE, TRANSBUF+6
SWPB  VARIABLE
      ○
      ○
      ○

```

FIGURE 28. TMS 9900 Assembly Language Transmission Formatting Example

predetermined size. It is the combination of these tasks into flexible length units of larger size (major frames) that determine frequencies of computation.

To complete the above definition, a description of volunteering is needed. As mentioned, CRMMFCS is an autonomous system of processors which reconfigure a predetermined set of tasks regularly. Because a central controller is not utilized, a distributed means of determining task selection is required. The solution was to have each processor, based upon its own view of its health, volunteer for tasks. Given the success of the volunteering and the current state of the SIM, a processor will be able to determine the next task to execute.

The volunteering process revolves around a SIM data structure called the volunteering table. This n-variable linear table (n is the number of processors) contains one of three types of data: cleared state, healthy state, or error state. The meaning of the value in each table location is dependant upon the time when the value is examined. Two valid types of time frames for the table exist: cleared table and volunteering. During the cleared state, the entire table should be cleared to a null state. This state is required to enforce refreshing of processors' volunteering status. During volunteering, any value may be in the table, however, only the healthy signal is accepted. All other values are interpreted as incorrect volunteering attempts.

Reconfiguration is accomplished with this table by using a pointer variable to indicate the logical start of the table. In the early stages of the project, the pointer was referred to as the Random Offset Pointer. Because the Random Offset Pointer was non-deterministic in nature, the pointer was altered to implement a Rotating Offset Pointer. This pointer moves in a known revolving manner around the table, overflowing from the end of the table, back to the beginning.



The pointer indicates which processor has the chance to claim the highest priority task. Following the orientation of the table, healthy processors are prioritized after this "first" processor. As the pointer moves around the table, the logical table ordering is changed, altering the task selection order; thus reconfiguring the system.

For a closer look at volunteering, refer to Figure 29a. After a volunteering cycle has taken place, the table will contain a number of locations with valid healthy signals (represented by a 1 in the example), and some with invalid or nonexistent signals (represented by 0). The rotating pointer will be pointing at one of the table locations, indicating the logical starting point. In this example, processor 7 will choose the highest priority task since it is the first healthy processor in the logical table ordering. Processors 8 and 9 do not select tasks since they are not healthy. As a result, processor 10 will take the second highest priority task, processor 1 the third, and so on.

Three subtask modules are utilized to accomplish the volunteering cycle in the current implementation. The first subtask performs the table clearing function. This subtask is performed near the beginning of each task. The second subtask begins with a self-test. If these tests are completed correctly, the processor volunteers with the healthy signal. The subtask is performed somewhere in the middle of the task set. The last subtask evaluates the volunteering table and determines the next task set to perform. The task set pointer corresponding to this is stored in the special RAM variable described in the kernel description above. This subtask is performed near the end of the task set. Because of interdependancies, the first subtask must be performed before the second, which is performed before the third. These subtasks are scheduled into all task sets in the system since the one

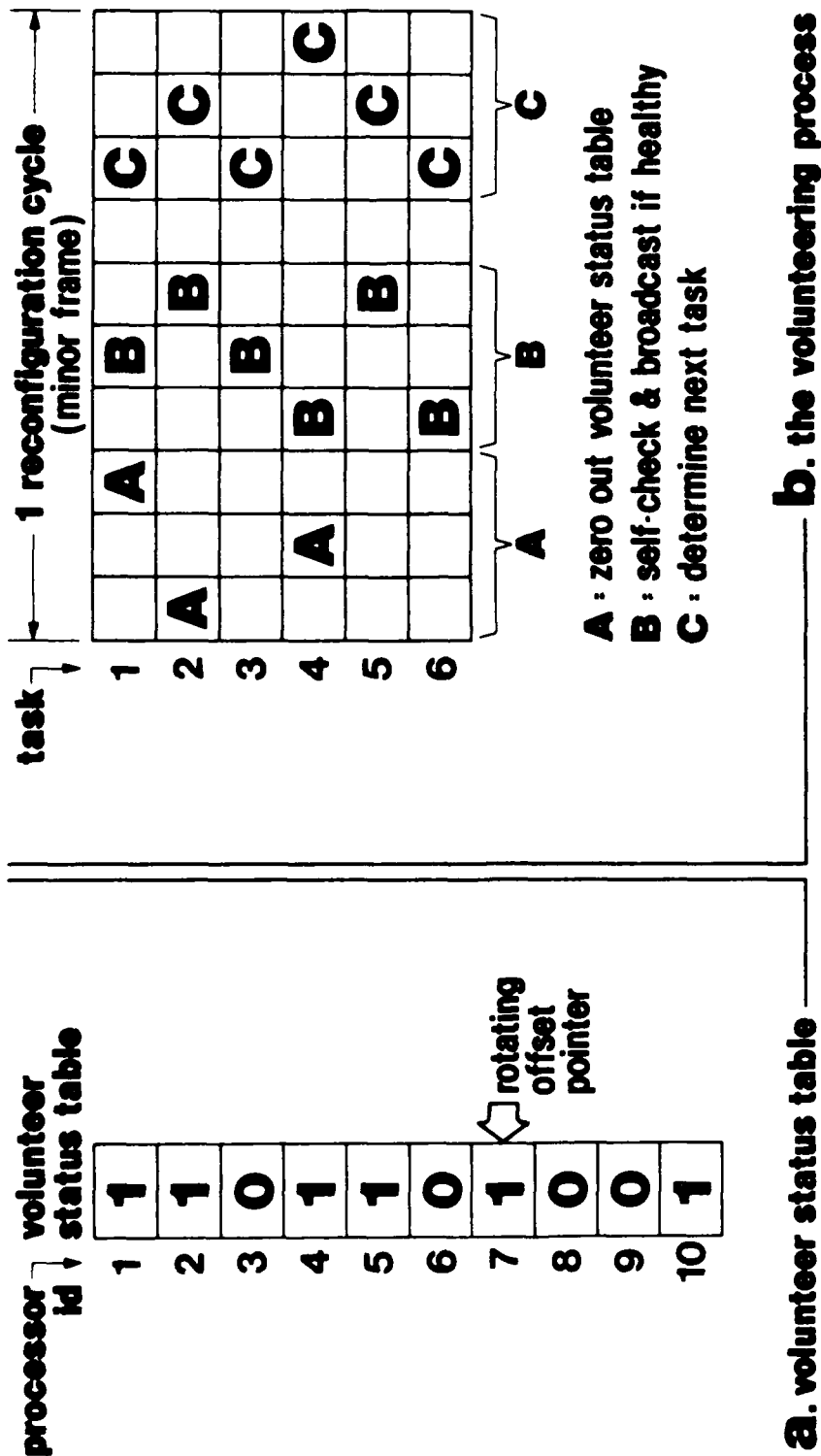


FIGURE 29. Volunteer Status Table and Process

function each task must do is to determine the next task in succession. As Section 5.1 will explain in more detail, this is not to imply that the overhead of CRMMFCS is 30% minimum (with 10 subtasks per task set). For simplicity, the software was structured this way for the laboratory implementation.

One of the key features of the volunteering and task tree system is the prioritization of tasks. Tasks are taken in order, with the processor whose volunteering table location is targeted by the rotating pointer getting the first chance to select. As a result, the first task set will be taken by the first healthy processor, the second task set by the second healthy processor, and so on. The  $n$  tasks will be chosen by  $n$  healthy processors. As processors fail, the unperformed tasks are the ones at the "bottom" of the tasks. This means that the first task set will be performed as long as there is one processor, the second set if two processors, and so on. Subtasks are placed within tasks so that the most critical subtasks are placed close to the "top" of the list of tasks. Tasks near the "bottom" are generally self-checks, which can be sacrificed. Other factors in subtask placement also come into play.

The description of how tasks can be organized using task assignment tables was covered in some detail in AFWAL-TR-81-3070 and will not be repeated. Instead, typical constraints and rules concerning subtask placement within tasks will be covered. However, these are rules-of-thumb given the application area of CRMMFCS and the goal of keeping the implementation as simple as possible. "Tricks" which can be used to bypass these rules and make an implementation more efficient are possible. Some of these will be addressed in Section 5.1.

One constraint already mentioned is volunteering overhead. The three volunteering-related subtasks are required for system operation. These three subtasks must be performed in order with at least one subtask time frame separating them (because of another constraint to follow). As described in AFWAL-TR-81-3070, these subtasks are usually kept in three distinct regions: subtask time frames 1-3, 4-7, and 8-10 respectively in the 10 subtask per task setup (see Figure 29b). Two of the three subtask types are in all tasks; the last is performed by a triplex subtask in the top three prioritized task sets.

Other overhead subtasks deal with failure detection and handling. These subtasks are flexible in their placement within a task assignment table and are the last subtasks entered in the table during task placement. The critical point in planning is priority placement. Most low priority tasks contain self-check subtasks. Other overhead subtasks, such as for SIM triplex variable error detection and general error reporting are of medium priority. Others are used to update system reconfiguration information (such as current task and mode information). These are given high priorities.

Application subtasks are merged in with overhead subtasks. Since the application is the real purpose for the system, related subtasks are appropriately prioritized. For fault tolerance reasons, application subtasks are executed identically by three different processors (in sets of triplexes). Since application subtasks require outputs from other application subtasks, these triplexes are scheduled close together to minimize the time to complete the "entire" computation (a computation is not complete until the three outputs of the triplex are completed). Since the triplex is spread over three task sets (no two of a triplex can appear in the same task since this would assume that one processor would handle two of the voted-on outputs), the

prioritization level is taken on the middle triplex subtask. A minimum of two outputs are required for a valid triplex vote. The third subtask of a triplex could be lost, and the system still operate on that function. If the middle subtask is also lost, the triplex output cannot be considered reliable.

An important detail concerning application subtasks is the time required for transmissions to reach their destinations in the SIM. The transmission of results from the previous triplex must be completed before future subtasks require them for computations. In the CRMMFCS laboratory implementation, one subtask time (one millisecond) is skipped between sequential triplexes (see Figure 30). A couple of factors cause this. First, transmissions are not sent by the transmitter until the next millisecond period after the buffer is loaded. Because of the uncertainty of buffer placement and contention, the programmer cannot be certain when, during the next millisecond period, the transmission will be sent or received. All transmissions during this period are effectively unavailable.

A method to avoid lower efficiency because of transmission delays is to interweave independent computations, filling the gaps that exist between application subtasks with overhead subtasks. Another way to attack the problem is to establish two independent equations and process them concurrently. The triplexes of these equations can be altered so that the intermediate transmissions of one occurs while the other is being computed.

A final factor in the scheduling of subtasks is the allotted time needed for handling transmissions. Currently transmissions require enough time to become a factor in considering subtask design and scheduling. The more transmissions a subtask must make the less time

variables in sim ready for use	<b>1</b>	<b>A, B</b>		<b>C</b>	
variables generated during this millisecond	<b>2</b>	<b>C = A + B</b>		<b>D = √C</b>	
variables in buffer ready for broadcast	<b>3</b>	<b>C</b>		<b>D</b>	
variables appearing on bus this millisecond	<b>4</b>		<b>C</b>		<b>D</b>
time (milliseconds)	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	

FIGURE 30. Required Latency of Scheduling of Subtasks

there is available for computations. Care must be taken in dividing functions into subtasks; the finer the division, the more intermediate transmissions required.

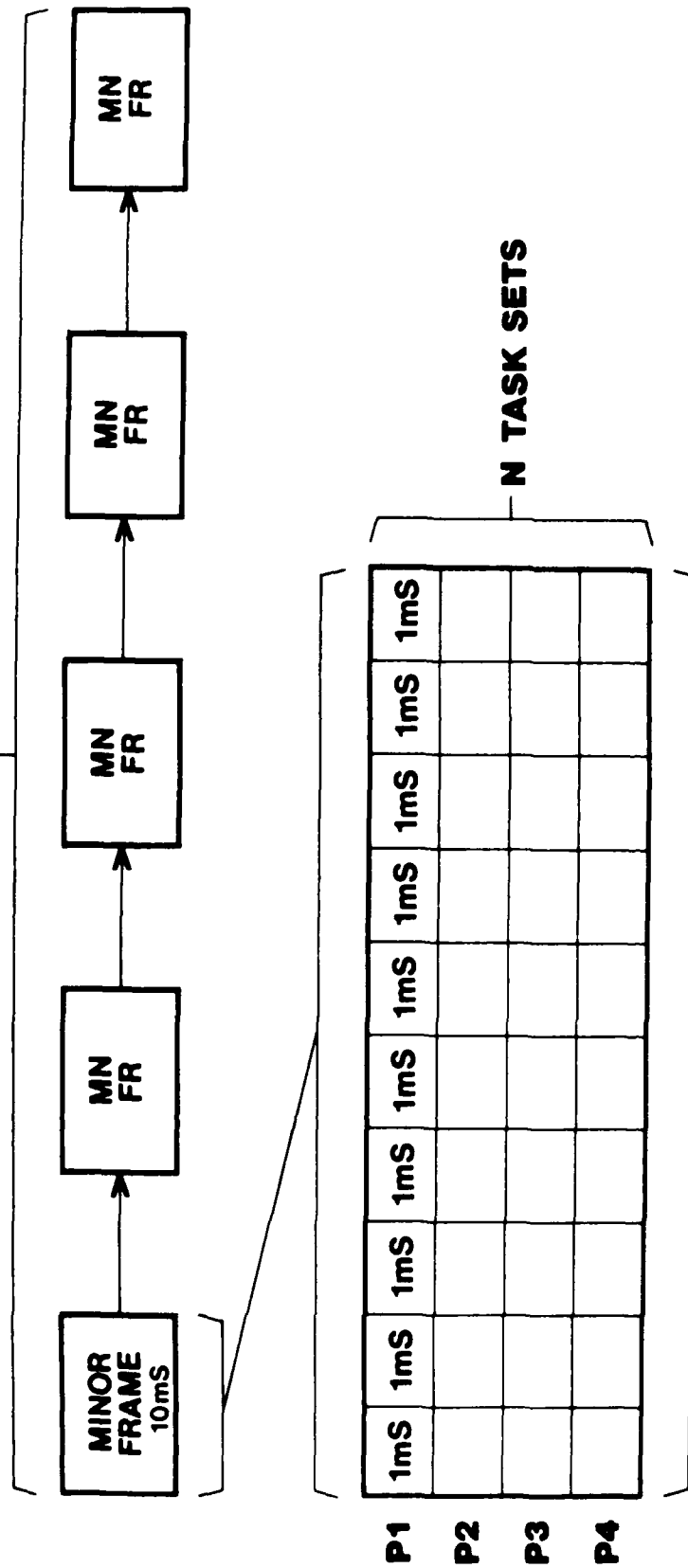
#### 4.2.5.7 Implementation Time Divisions

Up to this point, the discussion of subtasks, tasks, and modes has been kept generic. Subtasks have been designated one millisecond in length, and are called millimodules. Tasks are sequential orderings of ten subtasks (task sets). The list of tasks for the  $n$  processors make up a minor frame, or frame for short. A mode, which in the implementation is five minor frames in length, is also called a major frame. The major frame length is set to be the period of computation so that when executed iteratively, the desired sampling or output frequency required is easily obtained. In the laboratory implementation, the 5 frame major frame sets the model computation frequency at 20 Hz (see Figure 31). The choice of one millisecond for subtasks and 10 millimodules per frame is rather arbitrary. A variety of factors could cause any of these to be changed (this will be discussed more in the sections to follow). Only one of these time intervals, the interrupt-bounded millimodule, is hardware dependant; but even this could be changed if necessary. With the simple task tree structure, the software can be easily modified for different length time periods. In fact, within one implementation, several different length modes or frames can be utilized for various modes of operation.

#### 4.2.5.8 Fault Tolerance

One aspect which distinguishes CRMMFCS is its ability to operate reliably, even with failed components. The key to accomplishing this involves the ability to detect, isolate, and correct (or mask) faults in an allotted period of time. The first area, detection,

# MAJOR FRAME - 50mS



**TASK SET = 10 Millimodules**

FIGURE 31. Laboratory Implementation of Time Divisions



involves identifying problems through self-checks, triplex checks, hardware test circuits, and initiating the proper sequences to prevent the problems from affecting the system further. The process involves filtering the identified fault from the outputs of the system, and initiating the isolation of the cause of the fault. Isolation is often the most difficult part to accomplish. Faults may go through several stages, remaining undetected for a period of time. This type of fault which goes undetected for some period of time is often called a latent fault. Because these types of faults are so hard to isolate, they can cause serious damage to the system before any measures can be taken against them. As a result, isolation involves locating the fault quickly, or as a minimum precaution shielding the system from the effects of the latent fault, until it can be found and corrected. Correcting faults involves stopping the operation of the faulty circuits, or disconnecting them from the system. In either case, measures must be taken to make sure that the loss of the circuit will not adversely affect the system (such as making sure a spare takes over the task not being performed by the faulty processor).

CRMMFCS utilizes a variety of fault detection measures. No single method can be used to detect, isolate, and correct all errors that can be made. The idea in CRMMFCS is to use a variety of error "traps" whose coverage overlap one another to catch the widest spectrum of errors possible. This concept is known as the fault filter (see Figure 5), described previously in the Concepts section. The following section describes the software aspects which make up the filter.

CRMMFCS uses a two dimensional table in the SIM, called the blackmark table, to record processor error information. The rows of the table are one processor's record as seen by others. Columns are each processor's view of the others. Each entry is a count, or weight, of

errors. When one processor detects an error made by another, it updates its viewpoint of the other. See Figure 32 for an example of a state of the blackmark table. Although not implemented in the current version of CRMMFCS, a weighting system could be utilized to penalize a processor more heavily for more serious errors. Further study is required to determine an adequate version of such a scheme.

When errors are detected, they are not always reported immediately to the blackmark table. The reason is the time it takes to transmit data to the SIM. In order to insure that a subtask's time is not exceeded, the worst case transmission usage has to be considered. If errors were recorded directly, the maximum possible number of errors detectable in a subtask would need to be considered to account for possible transmission time. Since errors are assumed to be rare, this would amount to wasted time in the subtask period. To correct this problem, a temporary storage stack (called the blackball stack) is used to store error reporting information. When errors are detected, processors report them by storing the appropriate information on the stack. A special subtask is scheduled periodically to dump the stack information to the transmitter buffer (after formatting). The size of the stack and the frequency of the special subtask are really dependant on the predicted frequency of errors. In the implementation, an arbitrary size of five words for the stack and task frequency of once per major frame is used.

A primary method in detecting errors in the CRMMFCS system is the use of triplex subtasks. To use the triplex scheme, three processors must compute the same function from the same inputs. Each of the processors in the triplex produces an output which should match the outputs of the others. Each of the variables used by the triplex subtasks is a triplex variable; in other words, there are three copies

REPORTING PROCESSOR						
	1	2	3	4	5	6
1	0	0	0	12	0	0
2	50	10	34	40	27	52
3	0	0	0	23	0	0
4	0	2	0	17	0	0
5	0	0	0	48	0	0
9	0	0	0	31	0	0

FIGURE 32. Blackmark Table

of the variable in the SIM. When a triplex variable is used as an input to a subtask, the three copies are put through a triplex compare function. In the event the compare function finds that at least two of the three variables agree, the variable is labeled as valid and used in computations.

When a triplex error occurs (such as when one of three disagrees), the processor that commits the error is blackmarked by the detecting processor(s). The identification of the offending processor is made through the processor ID tag stored in the address word saved with the data in the SIM. In this way, isolation of triplex data errors is automatic with detection. To report on another processor, the detecting processor simply puts the offending processor's ID on the blackball stack. The next stack handling subtask will convert the ID(s) to the proper count(s) or weight value(s).

A total triplex disagreement (i.e., all of the triplex variables are different), in the current implementation, results in the current subtask computation being discontinued. The next subtask in the computation, which uses this subtask's outputs, will use the last iteration's value instead of an erroneous value. Past value usage in control systems is generally an acceptable option. Note that other methods for triplex disagreement fallback positions exist, such as the physical storage of past values; however, the method chosen was determined to be the simplest for the implementation.

Other types of checks also produce blackmark table reports. Self-check tests are an obvious example. Various types of self-checks are used in CRMMFCS. One type is the "spare" subtask as noted in Section 2. In the CRMMFCS implementation, two versions of this subtask type were produced. One performs a repetitive arithmetic computation which takes a seed number and attempts to produce a proper result. The

other performs an exhaustive processor status flag test which sets, clears, and tests all flags. The two subtasks were geared primarily towards testing the processor itself, not the associated communications hardware or memory. These additional tests could have been applied, but were not needed for the implementation.

Volunteering employs two types of self-checks. Before sending a volunteering transmission, each processor goes through a series of tests to determine its own health. The CRMMFCS implementation uses a seed number and a set of arithmetic and memory/register transfer operations to generate the proper BAG combination (all the operations required for transmission formatting are utilized). If any operation fails, the BAG will be disabled, and volunteering for the processor will fail. In addition, the proper volunteering status value, could be generated in the same manner. This was not done in the implementation.

The other volunteering self-test is volunteering itself. Volunteering tests a processor's basic ability to transmit values accurately, and is performed often enough to make it effective in detecting even intermittent transmitter problems. The volunteering process not only allows reconfiguration and autonomous control, but also helps to make the system more reliable.

Software implemented fault detection mechanisms are not complete in comprising the filter. Problems within the software or processor could cause the features to be bypassed or effectively ignored. Simple hardware test circuits can prove to be extremely useful in supporting such a filter concept. The CRMMFCS BAG circuit is probably the most important of the filter coverage areas. As described earlier in Section 4.1.7, the BAG is a lock and watchdog timer circuit which forces a processor to periodically refresh a port with a proper value in order to keep transmitter privileges. If the timer expires,

or the port is improperly loaded, the processor's transmitter is disabled. If the processor is unable to work the BAG correctly, it will subsequently be unable to volunteer for and take tasks or transmit data of any kind to the SIM. The processor is, then, effectively shut down.

In the current implementation, the watchdog timer of the BAG is set to 15 milliseconds (counts) every time the port is refreshed. This timer is decremented at each millisecond boundary. The initial count was chosen for convenience of hardware decoding in the implementation. For fault detection purposes, the count can be reduced, establishing the need to refresh the circuit more often. A tradeoff must be made. The more times the processor is forced to refresh the BAG, the more testing is performed, which verifies processor operation. Each time the BAG is refreshed, however, overhead is increased; especially if extensive key generation tests are performed each time. In the current implementation, the BAG is refreshed once per frame, during volunteering. As mentioned above, before a processor volunteers, it executes a self-test to determine its own health. The major portion of this test is geared towards generating the correct BAG key value. The outcome of the test is determined in the success of the volunteering attempt. An error made during the test will result in a bad key value, which will shut off the transmitter when loaded into the BAG and the volunteering transmission will never be sent. By working with the key generation tests and by shortening the watchdog timer period, the BAG can be made into an extremely powerful fault detection tool.

In order to support the above filter checks and reporting, special tasks are utilized in the CRMMFCS implementation. The first is a task which is assigned whenever a processor fails to volunteer correctly. In this case, since the processor has not proven its health, it cannot be assigned a task from the normal list. A special task is

assigned instead. The purpose of the task is to test the processor for a specified period of time to determine if that processor is healthy enough to return to normal processing (remember that bus errors, for example, can cause a good processor to fail a volunteering cycle; in this case it is not desirable to permanently remove that processor). In the implementation, the processor will hold itself in the special task for a period of 15 frames before resuming volunteering. During this time, a battery of tests will be run, including the BAG test once per frame. If the processor executes correctly during the suspended time, it will be permitted to retry volunteering to rejoin the rest of the system.

Another special task is executed whenever a processor determines that a majority of the other processors have recorded more blackmarks on it than is allowed. This comprises the blackballed state, in which a processor removes itself from the system. The BAG is disabled immediately, and the processor puts itself into a "do-nothing" state, from which there is only one escape. Because the condition that too many processors are blackballed out is possible (although remote), a means must exist to attempt to restart the system with at least partially operating processing units. Therefore, when the processor enters the blackballed state, it periodically reads the current SIM blackmark table state to see how many processors have disabled themselves. If this number exceeds the operational limit of CRMMFCS, the processor is given the opportunity to restart itself. This restart capability is built into another special task which buffers the operational state of CRMMFCS from the blackballed state. This task clears the previous blackballed processors "report card" in the blackmark table and resynchronizes for system operation. Volunteering is then allowed once again. Note that the BAG test and other barriers

must first be passed. This will ensure that processors with major failures will still be unable to rejoin the system. With this backup condition, the system will not necessarily go down in the case of too many processors blackballed. The need for this special condition will be discussed in a section to follow.

A special task is also assigned in the event a processor determines that its own volunteering table location is not cleared before volunteering takes place. The determination of this state is made during the pre-volunteering self-check described above. This condition could occur as a result of faulty SIM RAM, for example, which can make a processor determine that its table status value is always good, even if transmissions are not reaching that location. The cleared state is required of all volunteering table locations before a volunteering attempt can be made. The special task for this case performs, among other tests, a subtask which attempts to clear the volunteering table location affected. If unable to change the location to the cleared state, the processor stays out of the normal operation of the system.

This completes the description of the fault filter concept of CRMMFCS. The techniques described in this report are, of course, not complete for a full-blown flight control implementation. The steps taken in the laboratory were done for a minimum test case; to demonstrate what things can be done and what areas need further investigation. For further information into areas not covered in this study, refer to Section 6 on CRMMFCS Problems Identified.

#### 4.2.6 Application Software: the Control Law Model

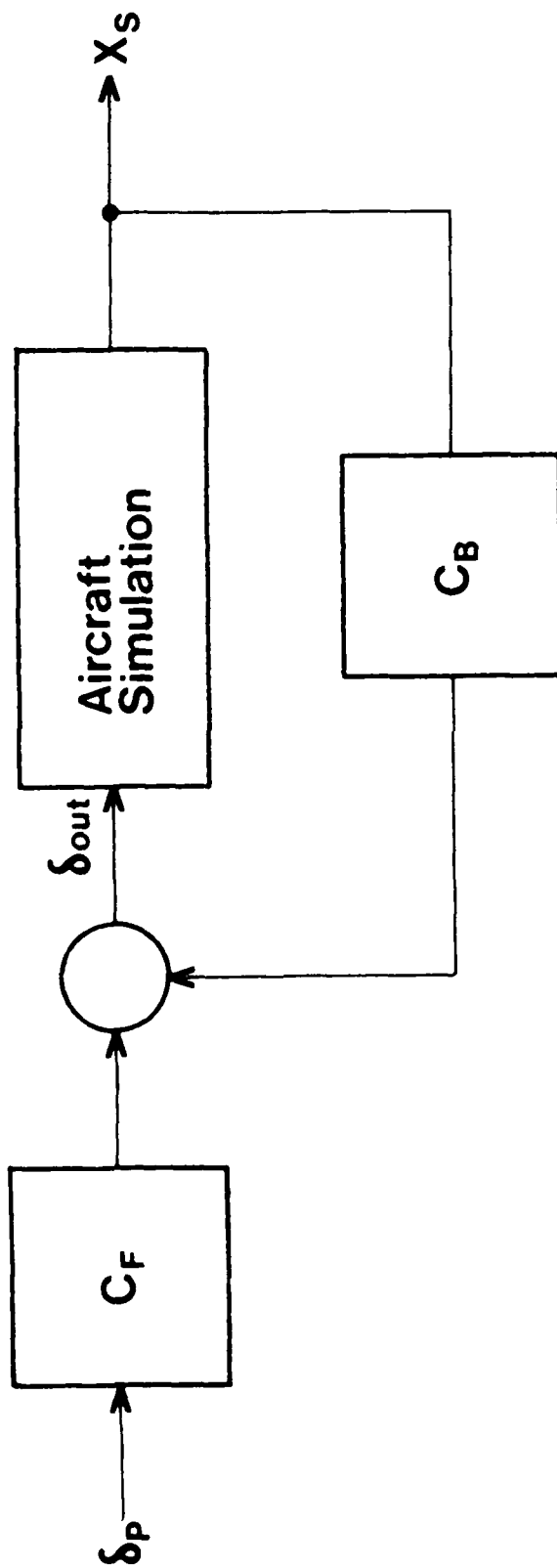
Up to this point, the focus of the discussion has been on the basic CRMMFCS functions. The description will now switch to the software domain, which is the main purpose of the system: the



application software. Given that the original purpose was to study and design a multi-microprocessor architecture for flight control, the natural selection for an application function was a flight control law equation. We decided to take only a portion of a full flight control model to keep the design and test of the software simplistic. A full flight control model requires much more time and effort, and the benefits of doing so for an architectural demonstration is not that significant. The main focus of the implementation was to demonstrate the reconfiguration and fault tolerance potential benefits. As such, a simple state-space model utilizing three inputs, three outputs, and four states was implemented to control the lateral motion of the aircraft model.

The model equations and progressive breakdown is given in Figures 33, 34, and 35. The purpose behind writing the equations out, then breaking them down, is to make the processing of the functions parallel. Intermediate SIM variables (Dlxx in the figures) are used since the final output variables cannot be computed within one subtask. Note that each equation in Figure 35 is processed in triplex, and requires a triplex output variable. Input variables are all in triplex.

Computations are performed in 16-bit scaled integer utilizing the 9900's on-chip multiply. The multiply instruction is unsigned with 32-bit results, so some extra code to account for signed values and rescaling must be included. Bits 0 to 14 and bit 32 are discarded (the initial values are converted to positive values for the unsigned multiply so bit 15 is always 0) from the result, and the rest is shifted into a singular 16-bit register. The sign of the result is determined, and the result readjusted as necessary. The constants were



$\delta_P$  = Pilot Inputs

$\delta_{out}$  = Control Surface Deflections

$X_S$  = Aircraft States

$C_F$  = Forward Control Coefficients Matrix

$C_B$  = Feedback Control Coefficients Matrix

$\delta_{out} = C_B X_S + C_F \delta_P$

FIGURE 33. Control Laws Block Diagram

$$\begin{bmatrix} D01 \\ D02 \\ D03 \end{bmatrix}_{3 \times 1} = \begin{bmatrix} C_B \end{bmatrix}_{3 \times 1} * \begin{bmatrix} X1 \\ X2 \\ X3 \\ X4 \end{bmatrix}_{4 \times 1} + \begin{bmatrix} C_F \end{bmatrix}_{3 \times 3} * \begin{bmatrix} DP1 \\ DP2 \\ DP3 \end{bmatrix}_{3 \times 1}$$

WHERE:

**D01** = RUDDER DEFLECTION

**D02** = SIDE FORCE

**D03** = AILERON DEFLECTION

**X1** = YAW RATE

**X2** = SIDE SLIP ANGLE

**X3** = ROLL RATE

**X4** = BANK ANGLE

**DP1** = PILOT RUDDER DEFLECTION

**DP2** = PILOT SIDE FORCE

**DP3** = PILOT AILERON DEFLECTION

FIGURE 34. Matrix Form of Control Equations..

$$\mathbf{D01} = \mathbf{DI11} + \mathbf{DI12}$$

$$\mathbf{D02} = \mathbf{DI21} + \mathbf{DI22}$$

$$\mathbf{D03} = \mathbf{DI31} + \mathbf{DI32}$$

$$\mathbf{DI}_{Y1} = \sum_{N=1}^4 \mathbf{CB}_{YN} * \mathbf{X}_N \quad \text{for } Y = 1, 2, 3$$

$$\mathbf{DI}_{Y2} = \sum_{N=1}^3 \mathbf{CF}_{YN} * \mathbf{DP}_N \quad \text{for } Y = 1, 2, 3$$

FIGURE 35. Separation of Control Law Elements

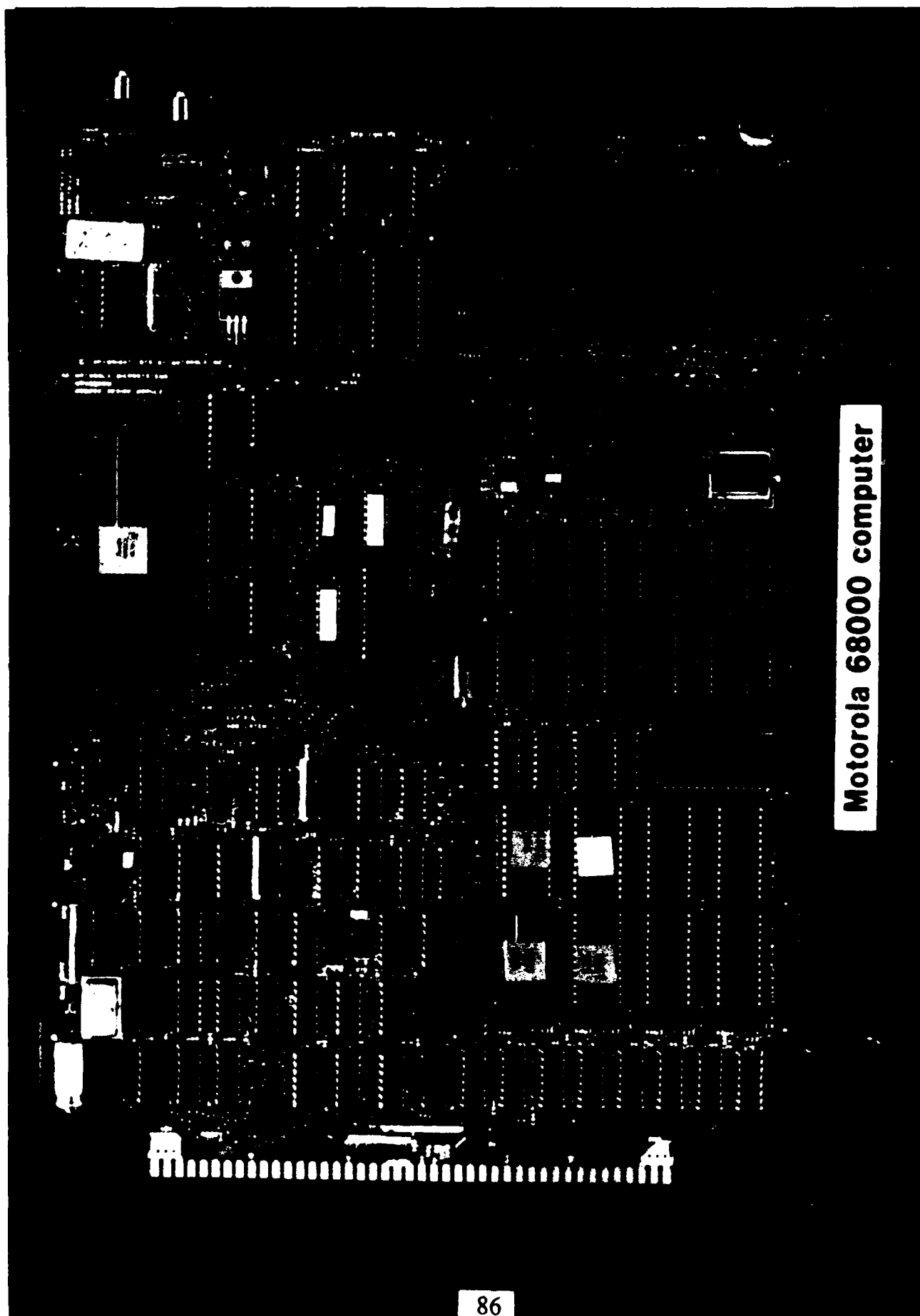
transformed from floating point to their integer value with scaling. Errors owing to the use of this scaled integer method proved insignificant.

Nine triplex subtasks are used to compute the model equations. These triplexes are kept in the three highest priority tasks to insure they are computed when operating a minimal system (often during demonstrations the system is taken down to two or three units through forced failures). As a result, the computations are spread over three frames. With an adequate number of processors and with proper scheduling, the time of computation could be reduced significantly. Since stability problems did not result because of the longer computational delay, and since the system was far from being strained functionally, the spreading of the subtasks does not seem to be a problem for this implementation.

#### 4.2.7 Simulation Software: MC68000 Airframe Model

Several options were considered for the airframe simulation for the above model. An analog computer was available for use; however, without a technique specified for I/O interface in CRMMFCS, this was not a favorable option. The basic assumption made at the conception of CRMMFCS was that smart I/O devices would interface to the CRMMFCS bus, possibly through some gateway. The control law processors of CRMMFCS would get their inputs through and send their outputs to the SIM. Since no formal definition of an analog interface was made, the analog computer option was discarded.

Given that the lateral motion model mentioned above and the associated airframe model were simplistic, the use of a digital computer for the simulation was constrained only by the method of communicating to the bus network. A MC68000 board (see Figure 36) was available for use, which was suitable for the computational tasks. At first, the



**Motorola 68000 computer**

FIGURE 36. MC 68000 Computer Board

easiest solution seemed to be a method of having the 68000 transfer data to a dedicated 9900 processing module connected to the busses. This did not turn out to be the case. Dedicating a 9900 processor to the I/O transfer task eliminates it from normal processing in CRMMFCS. Also, the means of transferring data at a suitable rate, and keeping the two different processors synchronized, turned out to be much more complicated than originally anticipated.

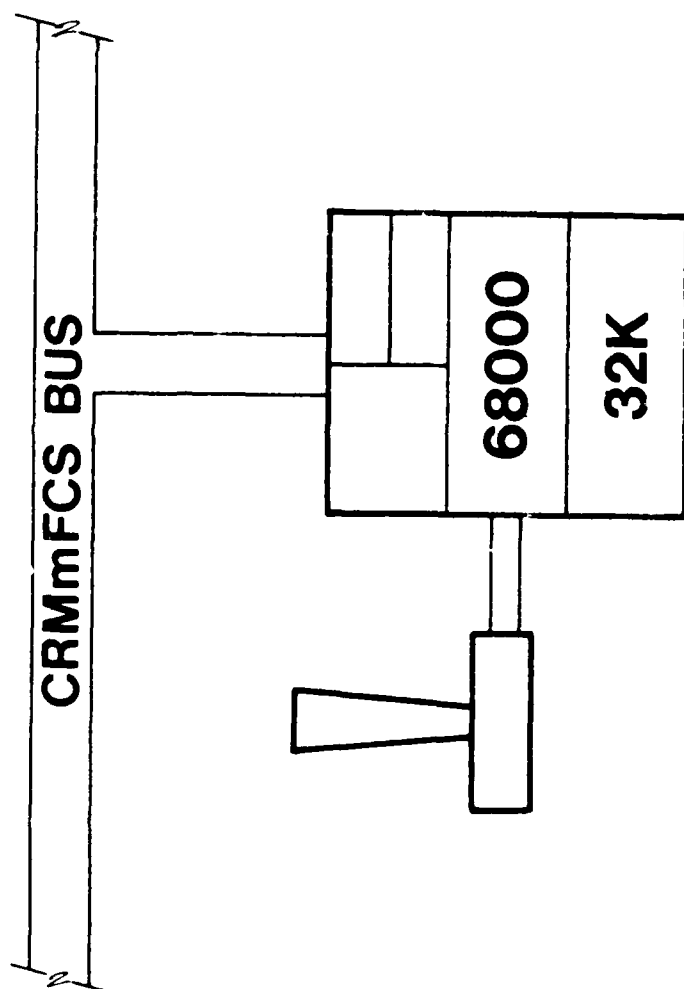
The adopted solution was the most simplistic (see Figure 37). Since CRMMFCS was not designed specifically for the 9900, adapting the 68000 to communicate directly to the transmitter/receiver hardware was not difficult. An adaptor board was developed to convert the 68000 backplane connector signals to match the 9900 card cage backplane connector signals. As a result, the 68000 board need only be programmed and plugged into a CRMMFCS card cage, as any of the 9900's would.

The equations used in the simulation are given in Figure 38. Since these equations are being computed by a single processor, no breakdown is required. The equations are computed at 10 times the rate of the control law model (or once every 5 milliseconds). Synchronization with the CRMMFCS processors in the implementation is done once at start-up, then is held through interrupts and software control, as in the CRMMFCS system.

The "pilot" stick input is also controlled through this processor. A simple video game style, switch-implemented joystick is sampled by the 68000 once every 50 milliseconds. The resulting step input value is sent to the SIM DP input variables.

#### 4.2.8 Other Software Developed

The 68000 simulation module is not the only non-CRMMFCS unit connected to the busses. Two displays are utilized in the CRMMFCS implementation, and these also use transmitter and receiver units to get



## AIRFRAME SIMULATION AND INPUT INTERFACE

FIGURE 37. MC 68000 Interface to CRMMFCS Buss..



$$\begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}_{4 \times 1} = F \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}_{4 \times 4} + \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix}_{4 \times 1} + G \begin{bmatrix} D_1 \\ D_2 \\ D_3 \end{bmatrix}_{3 \times 1}$$

FIGURE 38. Matrix Form of Airframe Simulation Equations..

at the SIM to monitor variables. One display monitors the flight control variables to give both a graphical and a raw data representation of the response of the system. The other display shows the major overhead data bases: the blackmark table and volunteering table. The overall operational status of the system can be monitored with these. Figure 39 shows the displays' layout in reference to the CRMMFCS bus network.

The control law response display is driven by a 6800 processor board connected to a transmitter/receiver pair in much the same way as the 68000 board above. Two different display outputs (switch selectable), are available. One display gives a graphical representation of an airplane, the sky, and the ground. The ground is hashed so that movement of the plane can be shown. The plane is held in the center of the screen and the hash marks on the ground are moved to simulate aircraft movement. The other 6800 display option gives the control law variables in raw decimal form. Both displays react in real time to SIM variable state changes.

The other display processor is a normal 9900 processing unit with an added 9918 graphics board developed in-house. This display also has two options which are switch selectable. The first display gives the blackmark and volunteering tables so that error updates and volunteering can be monitored (see Figure 40). This table is also updated in real time. The second option to this display shows which processors choose which tasks given the current volunteering state. In this way, reconfiguration of tasks can be monitored and demonstrated much more easily.

This 9918-driven display would not be as effective without some means of slowing the execution of the system down to a speed where the change of the variables could be watched. Real time execution

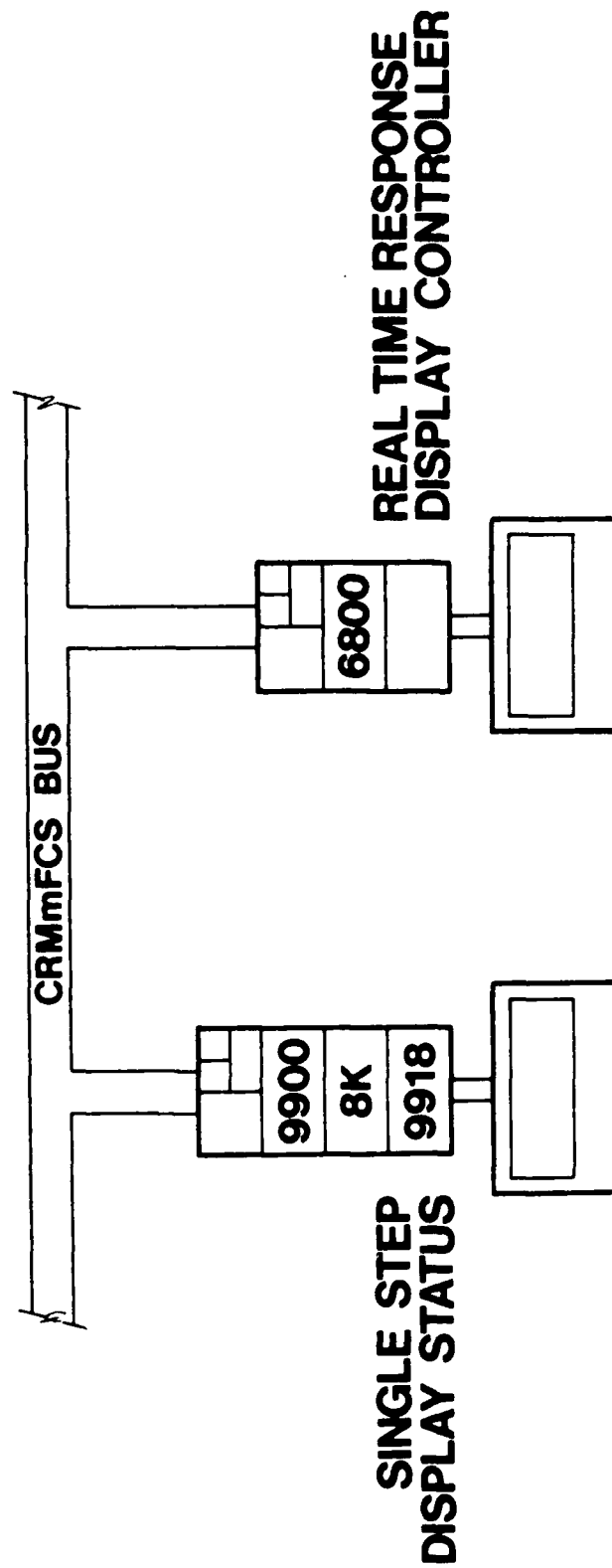


FIGURE 39. Laboratory Displays' Interface to CRMMFCS Buss

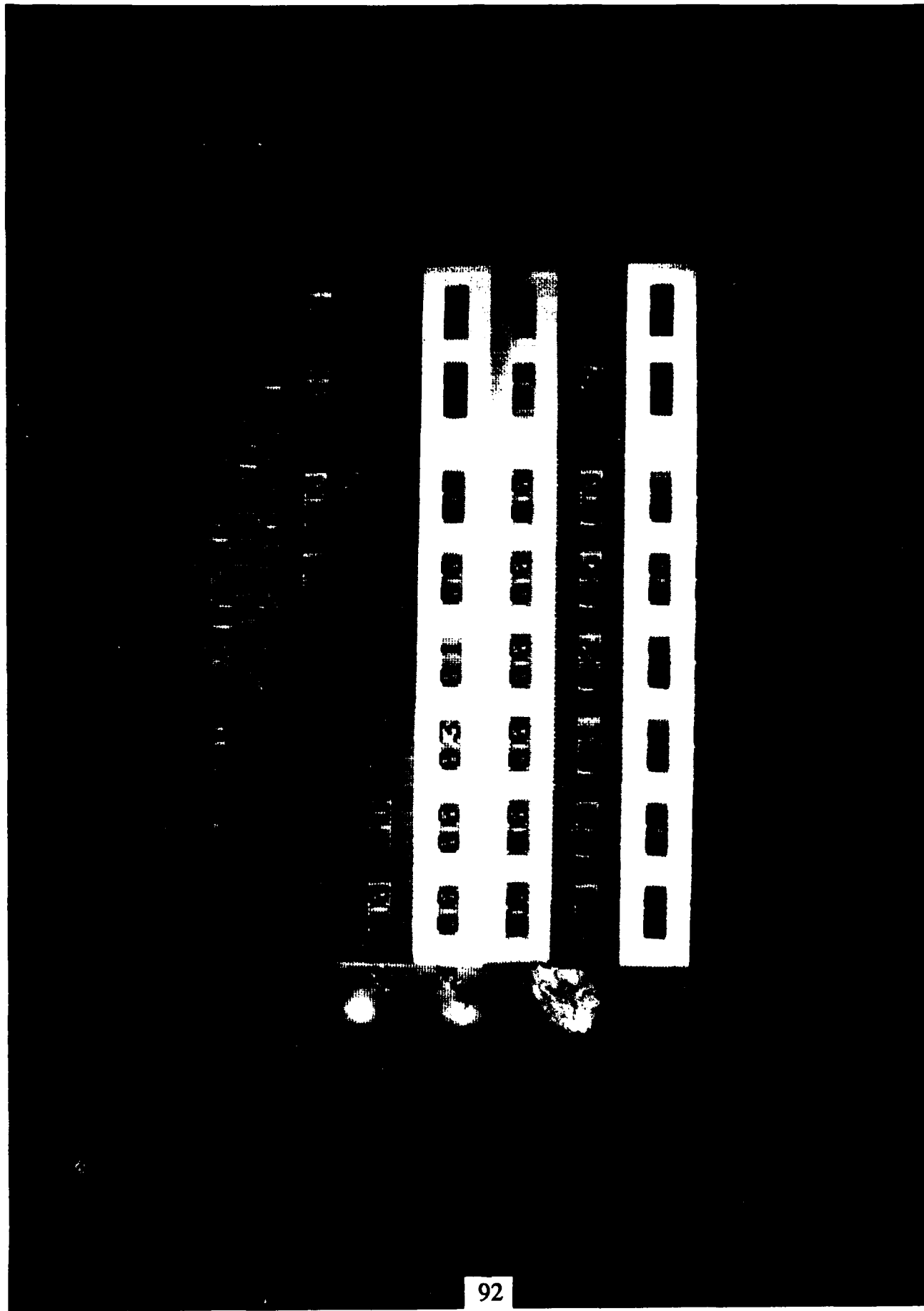


FIGURE 40. Volunteering and Blackmark Table

changes the variables quicker than a human can comprehend, so a single-step capability was added. This circuit was added to the bus termination hardware so that in "single-step" mode the millisecond sync pulses would be sent only upon the user hitting a single-step button. This process gives the effect of wait states on the system. The sync pulses drive the interrupts which allow subtasks to transition. Without the sync pulses, the kernels will wait before going on to the next subtask. As a result, the system can be single stepped, "millisecond by millisecond" during testing. The 9918 display can then be driven at a speed more agreeable to a user's comprehension.

The last in-house-developed software to be described in this report is the TRS-80 data collection software. A data collection circuit was developed in-house to capture transmissions off the bus, format them, and store them in RAM. After a collection of data is made, the RAM data can be transferred to the TRS-80 floppy disks for long term storage. This data is put through the software developed to give statistics of bus loading and processor-on-bus percentages. The software, developed in BASIC, was not used extensively in the implementation, so will not be described in detail. The results gained were primarily as expected: bus one was the most frequently used because of the bus access priority scheme set up in the transmitter, and bus loading was generally low because of the simplistic model being used. This software was developed primarily for later tests on varying bus priorities and more critical bus needs.

#### 4.3 Implementation Laboratory Demonstration

Figure 41 gives the implementation laboratory layout. A maximum of six CRMMFCS 9900 processing modules are run together at one time (owing to transmitter and receiver hardware availability). In addition, the 68000 airframe simulation processor, 6800 real-time

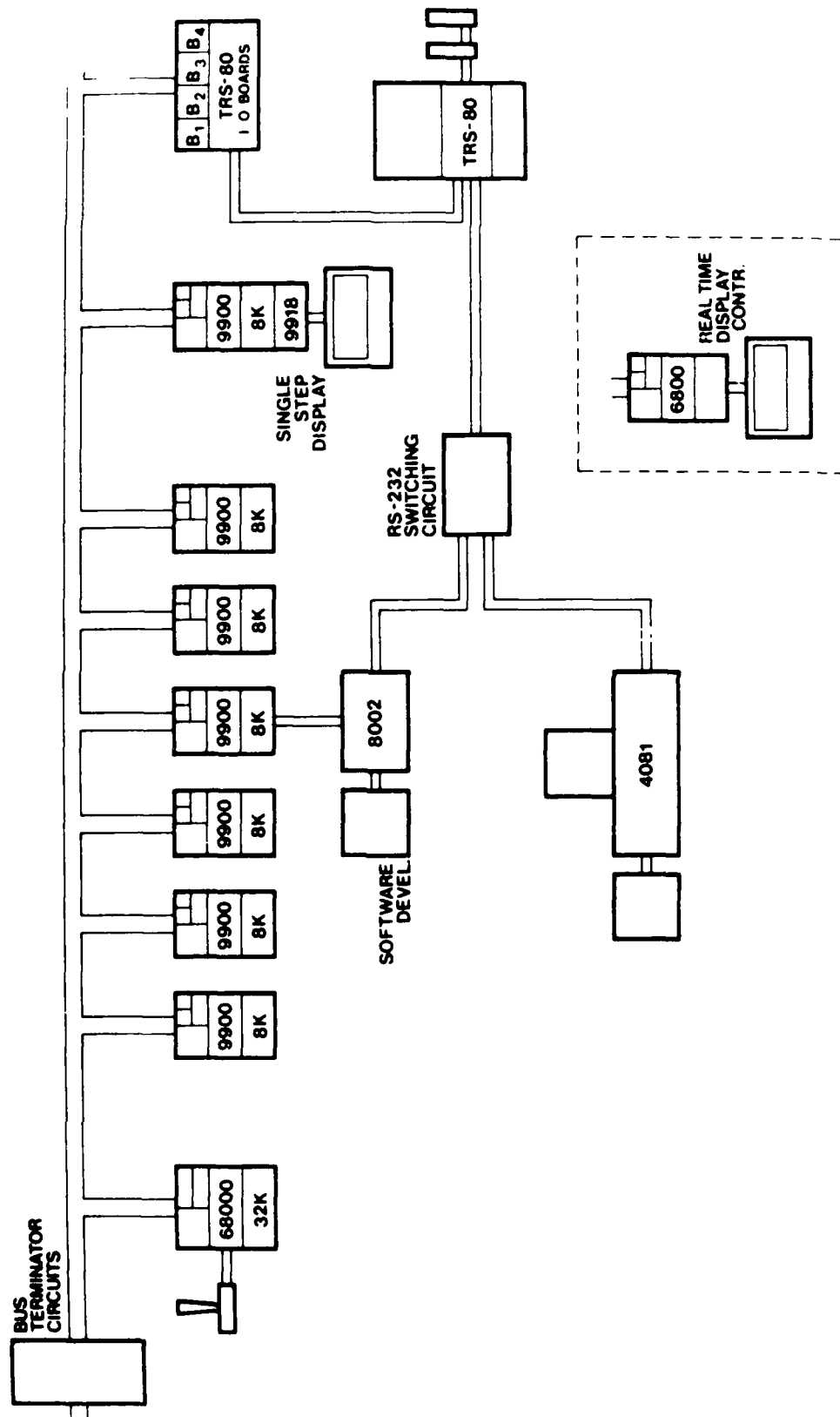


FIGURE 41. Systems' Laboratory Implementation

display, and 9900 status display are connected to the CRMMFCS network of busses. The bus termination circuit box, with its special "single-step" sync pulse switch, terminates the ends of the bus. The 8002 development system is connected to one of the processing modules for downloading and control. In the demonstration, these components faced the observation area. The joystick was attached to a "pilot's seat," in the middle of the demonstration area. During the demonstrations, one observer would man this seat to act as the "pilot."

After a description of CRMMFCS was given, the joystick would be used to demonstrate system response. Only left and right movements cause response with the model being used. The plane's apparent left and right directional movement on the 6800 display would then be observed under complete system operation. An explanation of the 9900 status display would also be given.

To demonstrate CRMMFCS's ability to withstand failures, various forced failures were induced in the demonstrations. A switch had been installed on one end of the bus to force a single bus low. The joystick would again be used to show continued system operation. Often, two busses would be "failed" to show further adaptability.

Another induced failure mode involved pulling off receiver connectors to the bus. This would completely isolate a processor from receiving transmissions. Volunteering fails immediately in this case, and the processor enters its self-test task. This is observed on the volunteering table on the status display, along with any blackmarks placed as a result. Often, more than one processor would be disconnected during a demonstration. After each removal, the joystick input would be reapplied to demonstrate continued operation. In extreme cases, all but two processors (the minimum case) would be removed.

Operation would still continue.

To demonstrate the operation of the BAG in stopping a processor, a switch was included on one transmitter to force the BAG output to the failed state. During the demonstrations, this switch would be thrown to simulate a BAG timeout. The associated processor would cease transmissions, and as such would fail volunteering. The status display again would be used to confirm the expected results.

Another switch was attached to the same transmitter to simulate a failed transmitter shift register bit. This "failure" would cause the transmitter to transmit wildly. The effects could be seen on the status display when some transmissions "hit" the wrong SIM addresses. Each time the switch was thrown, the associated processor would shut itself off as a result of fault filter detection.

## 5 Implementation Performance Data

### 5.1 Throughput

Generally, throughput determinations for systems are given in operations or instructions per second (OPS or IPS). However, many parallel processing architectures achieve high computation rates only through the use of extreme numbers of processors. Often, the overlooked factor is the percentage of the time an individual processor is able to compute application functions (i.e., the time a processor is not performing system overhead functions). This translates to processor utilization efficiency. In general, systems which have higher efficiency can achieve high computation rates at lower system hardware costs.

Although the TI9900 was chosen as the processor for the implementation, CRMMFCS is not dependant upon the usage of that processor. As mentioned previously, both 68000 and 6800 microprocessors are also used in conjunction with the transmitter and receiver hardware in the laboratory demonstration system. The backplane connectors used



by these the transmitters and receivers to communicate to the main processor were made to be TI9900 compatible, but this in no way limits CRMMFCS to 9900 microprocessor usage.

As a result, CRMMFCS is fairly independent of processor selection. Total system throughput will certainly increase as newer and faster processors are used; even though no significant architectural changes are made. As such, the following discussion will not give throughput figures in OPS or IPS. Instead, an indication of individual processor overhead will be given to show the efficiency of processor utilization.

First, consider the ideal parallel processing system. An ideal system of  $n$  processors would have zero percent overhead and, as such, would have a throughput of  $n$  times the throughput of one processor. This system would spend no time in selecting tasks. The bussing network would have to provide instantaneous data transfers, with no processor transmitter-usage overhead. In short, the processors would only compute application oriented functions. No system related tasks would be performed.

Of course, realistic systems do not have these characteristics. Overhead is needed to perform basic communications tasks and task selection. Fault tolerant systems must also perform self-checks and variable verification to ensure that errors do not go undetected. Busses have some transmission latency which may cause one processor or task to wait for the outputs of another. A realistic multiprocessor system must include some percentage of overhead along with the application functions.

In CRMMFCS, reconfiguration and fault detection are the primary contributors to overhead. In the current implementation, with 6 processors total in the system, 18 of 60 possible millimodules per minor

frame are used for reconfiguration (see Figure 42). Fifteen of these accomplish the three subtask functions for volunteering, as noted above in the Section 4.2.5.6. The remaining three are used to update the major and minor frame triplexes. Reconfiguration, therefore, contributes to 30% overhead in the current implementation.

This figure is not the minimum that can be achieved with CRMMFCS. Some of the overhead millimodules are somewhat less than the one millisecond maximum duration. As a result, the 18 millimodules can be combined to form a set of 12 millimodules which perform the same function (see Figure 43). The minimum reconfiguration overhead figure is then 20% for a system with 10 millimodules per frame (with 9900's). If the ratio of millimodules per frame were to be increased, this percentage could be reduced further.

The above discussion gives the overhead in CRMMFCS because of reconfiguration. Other factors can contribute to total overhead. The effect of these factors is generally determined by application requirements. Since the above reconfiguration subtasks are the only non-application oriented components of overhead required for system operation, the 20% figure is the minimum overhead percentage in a 10 millimodule per frame CRMMFCS set up. The aspects to be described below add to this.

Bus usage is a significant contributor. Each transmission in the current CRMMFCS implementation takes about 70 microseconds minimum to format and store in the transmitter buffer. The one millisecond transmission delay noted earlier in Section 4.2.5.6 can also contribute to overhead if independent threads of computation cannot be intermixed sufficiently. Reconfiguration can force higher bus usage by reshuffling tasks, thereby creating a higher data exchange requirement.

# MINOR FRAME

1	2	3	4	5	6	7	8	9	10
MM000			MM008	MM001				MM00A	
MM000			MM008	MM003				MM00A	
MM000			MM008	MM005				MM00A	
			MM008					MM00A	
			MM008					MM00A	
			MM008					MM00A	

NUMBER OF PROCESSORS..... 6  
 TOTAL NUMBER OF MILLIMODULES .... 60  
 TOTAL OVERHEAD MILLIMODULES .... 18  
 OVERHEAD UTILIZED ..... 18/60 = 30%  
 MM000 : CLEAR VOLUNTEERING TABLE  
 MM008 : VOLUNTEER  
 MM001, MM003, MM005 : UPDATE MAJOR, MINOR FRAME POINTERS  
 MM00A : DETERMINE NEXT TASK

FIGURE 42. Reconfiguration Overhead

MINOR FRAME									
1	2	3	4	5	6	7	8	9	10
			MM0081					MM00A0	
			MM0083					MM00A0	
			MM0085					MM00A0	
			MM008					MM00A	
			MM008					MM00A	
			MM008					MM00A	

NUMBER OF PROCESSORS ..... 6

TOTAL NUMBER OF MILLIMODULES ... 60

TOTAL OVERHEAD MILLIMODULES ..... 20

OVERHEAD UTILIZED ..... 12/60 = 20%

MM0081, MM0083, MM0085 : VOLUNTEER; UPDATE MAJOR, MINOR FRAME POINTERS

MM008 : VOLUNTEER

MM00A0 : DETERMINE NEXT TASK; CLEAR VOLUNTEERING TABLE

MM00A : DETERMINE NEXT TASK

FIGURE 43. Minimum Reconfiguration Overhead

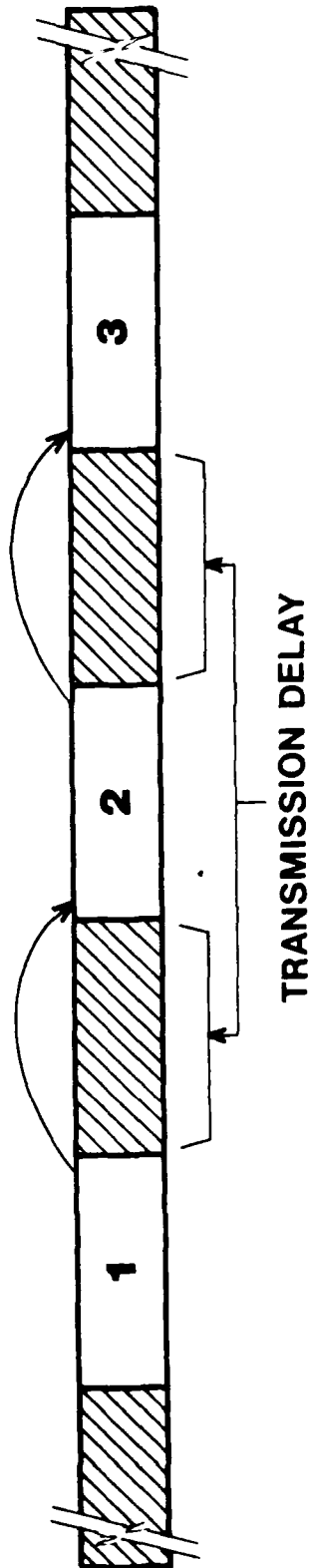
All these factors are determined primarily by the application. Each application has some number of variables to be computed. The means of computing the functions has various inter-relationships which determine potential parallelism. The complexity of the functions determines how much can be done by one processor in a certain time frame, thereby possibly creating a need for temporary variables and more transmissions.

Fault tolerance increases overhead also. Redundancy in computation and data storage tends to be the biggest component. In the CRMMFCS implementation, all application functions are performed in triplex. Although the three processors are operating in parallel, the effective throughput is of one processor. Triplex variables requires the use of triplex compare procedures to perform the voting on and picking of a good value. The reporting of erroneous processors to the blackmark table also falls under this particular category of overhead.

The application also determines the effect of fault tolerance on overhead. As the interchange of data between processors is increased, more triplex checking is required. As more checking is performed, more reporting is possible. If fault tolerance is not required by the application, none of this redundancy or checking is needed. CRMMFCS was conceived, however, as a fault tolerant system for flight control application. The cost of the extra overhead needs to be paid. Some tradeoffs can be made between fault tolerance and overhead considerations by adjusting the amount of interchange of data.

To reduce data exchange, a processor has to compute more before passing results. One means of accomplishing this is compound millimodules (see Figure 44). Compound millimodules are two or more regular millimodules in sequence, within a task, that pass intermediate variables between themselves through the local RAM of the processor, not

**(w/o) COMPOUND MILLIMODULES: Values passed via SIM**



**(WITH) COMPOUND MILLIMODULES: Values passed via local RAM  
(NO TRANSMISSION DELAY)**

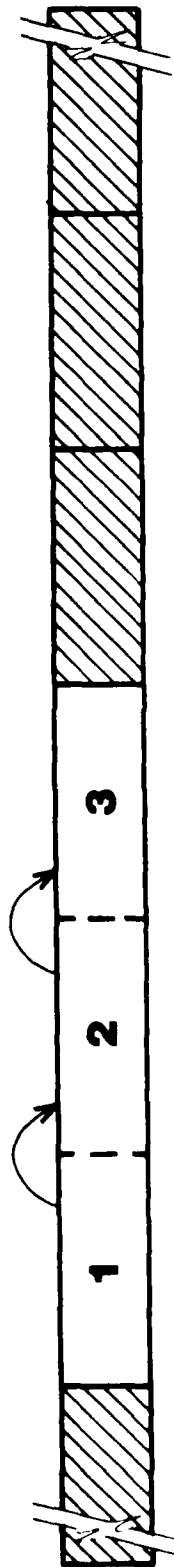


FIGURE 44. Compound Millimodules

through the SIM. Note that this definition is different from the compound millimodule definition given in AFWAL-TR-81-3070 report. This, in effect, creates a longer length millimodule that can compute more before having to pass results in transmissions to other processors. The reduction in transmission needs results in less transmission formatting, less waiting for transmissions to complete, and less triplex variable comparing. Synchronization and modular structure are still maintained, but a more efficient operating environment is available. The tradeoff was an increase in efficiency for a decrease in variable checking.

In general, all that can be said of the overhead of CRMMFCS is that it is greater than 20%. The amount greater than 20% varies from application to application. Consider the current CRMMFCS implementation with its 30% reconfiguration overhead percentage. Other overhead is required for redundancy (18 subtasks), blackball stack maintenance (6), and triplex error checking (9). With each application subtask transmitting a single variable as output, nine transmission formatting times, at about 90 microseconds each, need to be added (only 9 of 27 subtasks are considered since the other 18 are already considered triplex overhead). The loss of time amounts roughly to one subtask time. No transmission delays affect processor utilization. The net increase in overhead is 34 subtask times out of 300 possible, or 11%. The approximate overhead, then, in the current implementation is 41%. This translates to a processor utilization of 59%.

## 6 Problems Identified and Possibly Corrected

### 6.1 Corrections

In the course of designing and implementing a system such as CRMMFCS, problems are encountered which can change the way the concept was originally envisioned. In some cases, experience dictates better implementations. In other cases, criticisms make changes necessary

to meet current standards or conventions, or to fix oversights in the original design. This section will describe the changes made in the CRMMFCS concept since the previous technical report. The changes were possible because we identified the problems.

As noted above in Section 4.2.5.7, the major frame length was changed from 30 milliseconds to 50 milliseconds (3 to 5 frames) to better accomodate the iteration rate of the control function being computed. In the original CRMMFCS concept description, the three frame major frame size was chosen since this was determined to be most suitable for a wide range of computation frequencies. However, the tree task assignment table structure makes it easy to change this rate from application to application. Different rates can even be built easily into different modes of a single application by simply adding or subtracting descendants from the major frame tree node. There is no reason to have a standard minor to major frame ratio in CRMMFCS.

Another change noted above in Section 4.2.5.6 was the shift from Random to Rotating Offset Pointer for the volunteering table. The reason for the change is that a randomized pointer system is not deterministic. As such, verification of system operation through simulation or modeling is extremely difficult. The Rotating Pointer moves around the table in a deterministic, linear fashion. The implementation of this involves a simplex variable in the SIM. When the pointer overflows the end of the table or goes out of bounds, the pointer is reset to the beginning of the table. This change does not affect the performance of the system; continuous reconfiguration is still accomplished.

Probably the most critical of the problems identified in the CRMMFCS concept is the potential for SIM contamination. SIM contamination is the corruption of the SIM's of all processors with



incorrect data values. Shared memory systems without adequate protection mechanisms suffer from this problem. As long as multiple processors have open access to a section of shared memory, the chance exists that the memory can be corrupted by the failure of one of the processors. Although the following text demonstrates that, to some degree, solutions can be found, the problem is a serious one. Further research is required to prove that shared memory techniques can be applied in a highly fault tolerant system.

At first glance, SIM contamination seems to be a clear-cut problem. A processor or transmitter fails, and as a result, floods the SIM with bad data; crippling the system. However, the problem is neither this serious or this simple. Protection mechanisms and limitations of CRMMFCS can shield somewhat against the effects of what we will refer to as the "malicious processor" (a module which will attempt to contaminate the SIM with bad data). Additional software can also be added to detect such conditions, and compensate as needed. The following discussion will describe the issues in SIM contamination and will offer some solutions as implemented in the CRMMFCS demonstration system.

How can bad transmissions be sent to or placed in the SIM? Let us divide the possibilities into distinct cases, and examine the causes and effects. The four cases to be discussed are processor, transmitter, receiver, and bus related errors.

The case in which a processor is the cause of a bad transmission seems fairly obvious. A processor can move bad transmission data to the transmitter buffer for a variety of reasons. Hardware failures on the interface between boards could be the cause, but this could also affect memory, SIM, and BAG transfers. Software problems could cause a processor to either intermittantly transmit

incorrect values at correct transfer times (minor code problem), or transmit bad data at unpredictable times (major code problem). By simply not placing an EOF indication at the end of valid transmissions in the transmitter buffer the processor could cause the transmitter to attempt to transmit unwanted buffer contents. The effect is the same for all these cases: bad values from the transmitter buffer are transferred to the busses as correct transmissions by the transmitter hardware.

The transmitter can be the cause of a bad transmission if it takes data from the transmitter buffer and somehow changes some number of bits to form an incorrect transmission. An error of this type can be caused by a "stuck" shift register parallel input bit or a faulty bus driver. In any case, the transmission must be formatted (in terms of start, stop, and separation bits) correctly, or the transmission is invalidated at the receivers.

Receivers can be the cause of transmission errors if either the transceiver chips interpret the bus incorrectly or one of the receiver's shift register parallel output bits is "stuck." Note that "stuck" address bits can cause data to be placed incorrectly in the SIM. Stuck data bits cause bad values to be placed at the correct locations. Since the receiver checks the separation bits, any transceiver error must still have these bits intact. Note that receiver errors only affect that particular module's SIM copy. As such, these failures are not causes of true SIM contamination.

Bus errors can somewhat be eliminated in CRMMFCS. Noise on the bus will cause a miscompare at the transmitter of the sender (interpreted as a loss of contention), or will almost certainly corrupt the separation bits (since bus noise tends to come in bursts). A shorted or split bus is detected by the bus termination circuits.

Errors that occur during communications can, therefore, be attributed to either the transmitter or receiver, as described above.

Other CRMMFCS features limit the effect of malicious processors. Of these, the BAG circuit appears the most effective. The processor which is erroneously filling the transmitter buffer with bad transmission data continuously is one case where the BAG should work well. If a processor tries to execute a corrupted or nonexistent section of code, it will probably be unable to refresh the BAG properly. As a result, the malicious processor can be stopped within one BAG refresh period. The BAG will also terminate transmission capabilities if a bad key value is stored. This catches the case of a processor with memory transfer (to transmitter buffer, BAG, etc.) or address word computation problems. Note that the BAG key generation test performed during volunteering performs functions utilized in transmission formatting. If a processor cannot accurately generate address words in transmissions, it will also fail to generate a valid BAG key. In general, the BAG is most useful in stopping processors with major failure problems.

Another CRMMFCS safety feature which can be used to stop a processor which has entered corrupted or nonexistent code is the interrupt handler. As noted previously in Section 4.2.5.4, interrupts can only occur validly in the kernel. If a processor does not return control to the kernel before the interrupt occurs, the interrupt routine will stop the processor in a short, uninterruptable loop. Badly failed processors should, then, be caught in a millisecond or less.

Blackmarking is another way to stop a failed processor. If the processor stores its correct ID in with the erroneous transmissions, other processors will detect the problem and subsequently update the blackmark table report on that processor. As more damage occurs, more

reports will be made. If the processor gets to be too bad, the reports will overflow the allowable level, resulting in that processor shutting itself off. This entire process depends, of course, on the capability of the processor to use the blackmark table results to evaluate itself. Therefore, the blackmark table is really only useful in stopping processors that are only slightly failed.

One other consideration in the SIM contamination issue is the potential range of effect. In other words, how much SIM memory can one processing module affect, and how long will the damage continue? The first question depends somewhat upon the capability of the transmitters. As noted above, a transmitter can only transmit 21 transmissions per millisecond under ideal conditions. With contention, this figure will be lower. In software, a processor can really only format and store about 10 to 12 transmissions maximum per millisecond (with the 9900), if proper formatting procedures are followed and very little else is done. With 2K variables stored in the SIM, a malicious processor would need much time to corrupt a significant portion of the SIM. This fact seems to indicate that simple variable dispersion throughout the SIM would lessen contamination damage.

The second question has been answered in part above. The BAG, interrupt handler, and blackmark table can be used to detect and stop failed processors. Assume that the processing module manages to escape these traps for some period of time and continues, however limited, to transmit bad values. Assume also that the transmissions happen to localize on the SIM data structures in such a way that damage can occur. The main SIM data area types are the volunteering table, the blackmark table, and the application variables. How would these be affected?

The volunteering table is refreshed and reused once per frame, or ten milliseconds in this implementation. Any noncontinuous damage to

this structure will be overlaid by the next turn of volunteering. Even if a short "blast" affected a majority of the processors of the system, the result would only be a short transient period in which the affected processors would reconfirm their health, then proceed. Long term damage is a detectable condition. Backup conditions, such as using redundant, dispersed tables, are potential solutions.

Most application-oriented variables are also periodic in nature. Short term damage either would not adversely affect the triplexes (i.e., not affect more than one copy per triplex), would only create a transient condition where the computations would be frozen at the last valid past value, or would create a transient condition where false values are being output. The last case can only occur if the damage is such that the triplexes reach agreement with incorrect values. The second case will hold at a past value since the triplex compares will fail. Note that because outputs from one triplex application subtask are used as inputs to another and since sequential application subtasks are generally placed within a couple of milliseconds of each other, the "open window" between update and usage, during which the corruption can cause problems, is small. Note also that because the computations are periodic, when the damage ceases, new valid inputs will enter, good triplex values will work their way through, and proper operation will resume.

The blackmark table is the major area of concern in SIM contamination. The table is not redundant or periodic. If the area of corruption is large enough, innocent processors may start removing themselves from the system because of the erroneous blackmark reports. If too many processors remove themselves, the system becomes nonoperational. In this case, short or long term damage can be fatal. The solution must lie in software detection of this special case.

The solution reached in the CRMMFCS implementation was to have each processor that was blackmarked out to report, if possible, its failed status to a special failure table. As a part of the special trap task entered upon detection of being blackmarked out of the system, the processor will monitor this table; counting the number of processors that have failed. If too many processors fail and report to the failure table, each of the "failed" processors that detect the condition will get a chance to reenter the system. The processor's failure table entry and the blackmark table report will be cleared to allow reentry. The processor must first, however, pass several tests and be able to reenable the BAG. If all these steps are passed without error, volunteering is allowed. In this way, the system will only fail if all blackmarked processors are truly failed. Innocent processors can reclaim their healthy status.

As with the previous cases, the blackmark table corruption recovery scheme really only works if the damage is due to a short term malicious processor. If the damage were to occur for a "long period of time" (a couple of seconds or less in modern aircraft), serious consequences could result. Since the complex modeling and analysis of CRMMFCS has not been done, the fault filter's ability to protect the system has not been shown.

A probability analysis of a processing module being able to accomplish the above damage has also not been performed, mainly because of the number of factors involved in the calculation. The chances of one processor, in a random manner, damaging a data structure such as the blackmark table continuously, without falling into one of the above traps, is obviously remote. In order to get the quantity of data transmitted that is needed for serious damage, erroneous software would need to be at least a part of the cause. A single hardware failure in

the transmitter, as noted above, would leave the processor healthy enough to detect the problem and remove itself from the system. The probability, however, that existing, verified code could get modified (through memory failure, for example) in such a way seems negligible. To achieve such a disastrous situation, the code would almost have to be written specifically to cause the damage. From this, the terminology "malicious processor" was formed.

## 6.2 Other Identified Problems

Some problems have been identified with the CRMMFCS concept, with no solutions attempted. These problems, like the SIM contamination issue above, were by-products of the way the concept was formed to meet other requirements. The first two problems to be discussed are not issues in the current implementation. Only if the system is expanded to meet greater tasks will these problems surface.

The blackmark table, as noted previously, is a two-dimensional table which gives each processor's view of the others. As such, the table size is  $n \times 2$ , where  $n$  is the number of processors in the system. As the number of processors increases, the table size expands at a much higher rate. The table can become unmanageable and consume considerable SIM space if many processors are added to the system.

SIM size, in general, is a significant concern. The current implementation has a SIM with a 2K variable potential. The question is whether this is a sufficient figure for a real flight control system. Varying modes, for example, could require different sets of state variables. The blackmark table, as mentioned above, takes up significant space if enough processors are added. All processors have to have access to all the data for the system of autonomous processors to execute. Certainly, some applications will require more than 2K variables.

Adding more memory to each receiver board for the SIM is possible. If this happens, however, the transmission (address word) size will have to increase (among other things). The ramifications, such as real estate increases, from such a change have to be weighed carefully.

Other than the BAG, all fault filter techniques in CRMMFCS depend upon the individual processor's ability to recognize failures or the indication by other processors that it has failed. If recognition is accomplished, the processor can shut itself off so that the effects of the failure do not spread. If the processor cannot recognize the failure and the BAG does not stop the processor, bad transmissions can be made which can have an effect on the entire system. The question that results is: by what means other than self-diagnosis or the BAG can a faulty processor be shut down? In CRMMFCS, one processor can only communicate with another through the SIM. No physical means exists for processors to shut off others; otherwise a problem more serious and complex than SIM contamination could result. Still, some means other than self-diagnosis is needed for processor removal.

#### 7 Updates to CRMMFCS if a Redesign was Attempted

As the implementation of CRMMFCS proceeded, potential improvements to the design were identified. Some of the software modifications to solve identified problems were described in a previous section. This section will describe those aspects which could have been changed or added; but were not, because of the extent of modifications required. Some of these improvements were technology upgrades. Others were architectural changes. All are features that would probably be included if a redesign of the CRMMFCS implementation were attempted.

The goal of the CRMMFCS implementation was to demonstrate and test the major concepts and demonstrate the potential reliability; not



to construct a state-of-the-art computing system. As such, the design of the system did not include the fastest processors available, or use the highest clock speeds possible. Many features of CRMMFCS could be redesigned to operate at higher speeds. A case in point is the use of a MC68000 microprocessor with a transmitter/receiver pair for the airframe simulation. This microprocessor could have easily been used as the main computing engine in each processing module. The use of a higher speed processor, perhaps with a floating point coprocessor, could have allowed a far more impressive demonstration application. The busses, as implemented, could have had their transmission clocks increased to two or three times their current speed. Faster and more highly dense memories could have been used. All these aspects could have been implemented, but their impact would not have better proven the concepts.

Not all the potential improvements are technological, however. Some changes can be made to correct features that have proven to be undesirable. The width of the transmitter buffer is one example. Although the 16-bit TI9900 was chosen as the processor for the implementation, the buffer width was designed to be 8 bits. This mismatch creates a need for extra software to swap and store the individual bytes of the 9900 16-bit word. This extra code can be awkward and bothersome to the programmer, and forces the storage of a transmission to take almost twice as much time. The disadvantage of the change is that it requires significant modifications to the transmitter hardware and additional board real estate.

Another undesirable feature of CRMMFCS that could be changed is total software control over the storage of the processor ID in transmissions. This implementation creates a potentially troublesome situation where a processor stores an incorrect ID in its transmissions. This can prevent other processors from blackmarking out a faulty

processor, and can even create a situation where an innocent processor is blamed for another's error. The correction to this seems simple enough. The software can still store the ID; however, before the transmitter sends the data, the stored software ID is compared against the hardware ID (perhaps a redundant copy). If the IDs disagree, the transmission is discarded. This could be another useful guard against the SIM contamination problem, discussed in the previous sections, since it forces the processor to at least partially format the address word accurately. Again, the reason this change was not included in the current implementation was the scale of the modifications.

Another useful change to the transmitter buffer circuitry involves clearing out the buffer page after it is dumped to the busses. This action has two favorable effects: a page can never be transmitted more than once (which can happen now if the processor does not reuse or clear the buffer page), and the EOB is automatically restored. Both situations could result in erroneous data being sent to the SIM. One simple implementation involves holding the buffer memory address after the read, then writing a hard-wired zero to the same location once the transmission is completed.

Given the quantity of discussion in this report on SIM contamination, an obvious modification is some sort of protection mechanism for the SIM. The protection can be accomplished by limiting access to the SIM. Many implementation possibilities exist. One technique (often used in single CPU, multiprogramming computers) is to segment (or page) the memory and apply a key and lock system on each segment. The segments in CRMMFCS could be unlocked by time of access. In other words, the software can be constructed so that certain areas of the SIM are only validly accessed during specified times. These sections could be locked by the receiver hardware during the other

times. The keys also could be associated to tasks. In this case, only certain tasks can access certain SIM sections. Keys are then selected with tasks by the processors. Implementations like these are far more complex to design and construct, but provide a powerful error-propagation prevention capability.

More comprehensive self-check subtasks could be added. The current self-check subtasks limit their scope to the processor itself. To be truly effective in detecting module failures, tests of transmitter and receiver functions should also be performed while on-line. Two obvious checks which can be added are a test of the BAG lock mechanism and a transmission verification test. The latter is performed to some degree with volunteering, but a more complete check-out is desirable.

## 8 Further Areas of Investigation

### 8.1 Benefits of CRMMFCS and Related Research

Until recently, multiprocessor architectures have been utilized in few real applications. Most development efforts have been limited to laboratory investigation systems, such as CRMMFCS. These efforts have demonstrated the vast potential of today's research and technology. The area is still developing, and problems remain to be solved.

Implementations like CRMMFCS have shown, however, that the time is approaching when multiprocessor architectures will be used for a wide variety of applications; including fault tolerant control.

Flight control (and, in general, ultra-reliable) systems have traditionally been developed as dual, triplex, or quad channel systems. These channels are solely redundant computing systems: each computes the same as the others, and the results are voted on. No true parallelism exists at the channel level. Channels can be comprised of one or more processing units.

Often, single CPU channels are used. In this case, the CPU tends to be a powerful processing unit. The additional computational power carries costs, however. The "black boxes" contain processing elements, memory, and I/O sections comprising several boards each. These single CPU channels suffer from the so-called "Von Neumann bottleneck": single processing element systems reach a technological limit in speed increases.

Multiple CPU channels can be constructed for speed increases when needed. This type of architecture has two benefits: the throughput potential to meet the ever increasing needs of systems, and the experience and proven reliability of "channeled systems." The cost in hardware may outweigh these gains. A triple channel system has three processors, or processor networks, dedicated towards a single task. In effect, only one in three processors is computing; the others are checking the answer. A 4 CPU per channel system in a triplex channel configuration, therefore, requires 12 processors to do the work of 4.

The introduction of CRMMFCS and other similar research programs have demonstrated that multiple processor systems in a "non-channeled" approach can be used to achieve both high throughput and high reliability, without the extreme penalty of dedicated redundant hardware. In addition, other benefits have been shown. Distribution of processing elements is possible to help protect the system in a hostile environment. Reconfiguration of tasks or control algorithms in such a case can result in a battle damage survivable system. With the use of pooled spares, the system can be made to degrade gracefully; that is, lose processors over a long period of time with minimal system performance degradation. The result is a system which operates longer between maintenance actions.

## 8.2 Other Issues:

CRMMFCS, and other efforts in the area to date, are not the complete answer, though. The sophistication and complexity of avionics and flight control systems is growing rapidly. Aircraft are continually being designed to fly and maneuver at higher levels of performance. Higher sampling rates from sensors, more complex control strategies and algorithms, and the use of High Order Languages (among other things) will create a need for higher speed computing. As noted in the previous technical report: "... the only thing growing faster than computer technology is the size of the problems to be solved ... ."

Other areas need consideration when applying parallel processing to the flight control domain. I/O integration was one area not given much thought when CRMMFCS was conceived and implemented. This area is as critical as the flight control "computer" itself, however. If the transfer of information of data to/from actuators/sensors is unreliable, relative to the flight control system, the goal of ultra-reliable control is diminished. The expansion of the reliability boundaries to cover I/O requires serious attention.

Programmability is another critical issue. Parallel processing inherently causes the software to be more complex. If individual processors in a multiprocessor configuration are programmed separately, for example, a substantial development and testing process could result. Few programming languages support parallel processing directly, and even fewer compilers assist in the construction of efficient parallel tasks. The enforced usage of ADA in DoD programs adds more design constraints, and possible timing difficulties. The use of parallel processing in flight control is, therefore, dependent upon careful consideration of programming issues.

Expandability, modularity, and adherence to standards are other factors in advanced research in this domain. Expandability is desirable to meet growing problems and the addition of new functions. Modularity can decrease costs by reducing hardware design requirements and allowing interchangeable parts. Standards are being (or have been) developed for areas such as High Order Languages, Instruction Set Architectures, and High Speed Data Busses. Research into the advantages and problems of these standards in relation to parallel processing flight control systems is necessary in order to develop advanced systems.

Investigations into reliable, multiprocessor control architectures are currently being performed by WRDC/FIGL as a continuation of efforts begun in CRMMFCS. These efforts are a part of the Flight Control Computation and Systems Integration Technology (FCCSIT) program which was started after the conclusion of the CRMMFCS work. In general, this program continues the push to advance the usage of microprocessors, digital technology, and advanced multiprocessor architectures and software in flight control and related areas.

## 9 Conclusion

This concludes the report on the implementation and analysis of the Continuously Reconfiguring Multi-Microprocessor Flight Control System. A demonstration system has been successfully built, programmed, and tested. Concepts have been shown, problems have been identified, and ideas for the future have been developed. The demonstration system still operates in the laboratories of AFWAL/FIGL, but this does not represent the end of work in the area. Even as this report is being written, new concepts and hardware are being developed for the next generation work in highly reliable multiprocessor architectures for flight control.

To achieve any goal, any advance in technology, incremental steps must be taken. Background investigations are performed. Ideas are tried. Tests are run. The work involved in CRMMFCS and other related programs combine to form another step along the path towards highly reliable, high performance aircraft systems.

## BIBLIOGRAPHY

1. S.J. Larimer and S.L. Maher, "A Solution to Bus Contention in a System of Autonomous Microprocessors," Proceedings of the IEEE 1981 National Aerospace and Electronics Conference, May 1981, pp. 309-317.
2. S.L. Maher and S.J. Larimer, "Continuous Reconfiguration in a Multi-Microprocessor Flight System," AGARD Avionics Panel on Tactical Airborne Distributed Computing and Networks, June 1981.
3. W. A. Crossgrove and L. A. Smith, "Distributed Systems: The Next Integration Method," AIAA 2nd Digital Avionics Systems Conference, November 1977.
4. J. A. White, et al., A Multi-Microprocessor Flight Control System, Honeywell Interim Report, August 1980.
5. S. J. Larimer, "Managing Software in a Continuously Reconfiguring Multi-Microprocessor System," Proceedings of the 1981 Joint Automatic Control Conference, June 1981.
6. The Engineering Staff of Texas Instruments Incorporated Semiconductor Group, "TM 990/100M Microcomputer User's Guide," October 1979.
7. Motorola Semiconductor Products Incorporated, "MC68000 Systems Design Module User's Guide," Third Edition Copyright 1980.
8. National Semiconductor Corporation, "DS3662 Quad High Speed Trapezoidal Bus Transceiver Data Sheet," February 1981.